Evaluating and Refining Diagrams that Support the Comprehension of Concurrency and Synchronization

by

Shaohua Xie

(Under the direction of Eileen T. Kraemer)

Abstract

Software systems that employ concurrency (e.g. online reservation, e-commerce, banking, brokerage, social networking, and medical systems) are becoming more and more prevalent. For such systems, the safe use of concurrency and synchronization and its optimization are key to success. However, practitioners often find multi-threaded concurrent software systems difficult to design, verify and maintain. Common difficulties inherent in concurrent software include safety, liveness, and non-determinism issues.

Our research aims to investigate which aspects of learning about and employing concurrency and synchronization are most difficult and what methods and tools can be used to effectively ameliorate the difficulties. Specifically, this research (1) empirically investigates the challenges encountered by novices engaged in learning about concurrency; (2) empirically evaluates the usability of existing notations, tools, and methods intended to address those challenges; (3) proposes, develops and empirically evaluates new notations and tools; (4) generates a body of reproducible, empirical knowledge that informs improvements in teaching proper use of concurrency and synchronization.

Index words: UML, Empirical studies, Software engineering notations, Concurrency and synchronization

Evaluating and Refining Diagrams that Support the Comprehension of Concurrency and Synchronization

by

Shaohua Xie

B.E., Wuhan University, Wuhan, China, 2001

M.B.A., Troy University, Troy, AL, 2003

A Dissertation Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2008

Evaluating and Refining Diagrams that Support the Comprehension of Concurrency and Synchronization

by

Shaohua Xie

Approved:

Major Professor: Eileen T. Kraemer

Committee: Krzysztof J. Kochut
Maria Hybinette

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2008

Acknowledgments

It is not possible for me to succeed at the completion of my dissertation, which marks a milestone in my life journey, without the support of many people. I would like to express my sincere gratitude to the following people.

First of all, I thank Eileen T. Kraemer, my major advisor, who has always been very supportive and sapient in guiding me throughout my doctoral study. I thank Krzysztof J. Kochut and Maria Hybinette for offering valuable insights into my research. I would like to thank my fellow researchers at Michigan State University: R. E. K. Stirewalt, Laura K. Dillon, Scott D. Fleming, and Yi Huang. Their collaboration has helped me to make consistent progress in my research.

I also greatly appreciate the help with my research and study from the faculty members in the Computer Science department: David K. Lowenthal, Shelby H. Funk, Liming Cai, Suchendra M. Bhandarkar, John A. Miller, and Ismailcem Budak Arpinar. I thank Jan Mrázek for offering me a research assistantship in his lab.

Thank you to my fellow students—Jianxia Chen, Tianhao He, Ting Yang, Xingzhi Luo, Shanshan Ding, Zhengyu Zhong, Philippa Rhodes, Kelly Storm, Shradha Kaldate, Manish Agarwal, Dongsheng Che, Maciej Janik, Haiming Wang, Hui Wang, and many others—for making my life enjoyable in Athens, GA.

I am deeply grateful to my family for their non-interruptible love and unconditional support. I thank Yinqin He for her encouragement and faith.

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Software systems that employ concurrency (e.g., online reservation, e-commerce, banking, brokerage, social networking, and medical systems) are becoming more and more prevalent. For such systems, the safe use of concurrency and synchronization and its optimization are key to success. However, practitioners often find multi-threaded concurrent software systems difficult to design, verify and maintain. Common difficulties inherent in concurrent software include safety, liveness, and non-determinism issues. It stands to reason that many of the difficulties related to concurrency may derive from the challenges that make concurrency and synchronization difficult to learn in the first place.

Moreover, research in computer-science education suggests that concurrency and synchronization concepts are generally difficult to master [7, 8, 14, 26, 37, 50]. The comprehension of concurrency 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 in terms of low-level synchronization primitives, which synchronize threads indirectly by modifying unseen operating-system-level data structures such as condition queues and mutex locks.

In a typical pedagogical setting, instructors usually introduce these concepts using small model problems—e.g., producers and consumers and dining philosophers—where the solutions employ operating-system-level synchronization primitives, such as *lock acquire*, *lock release*, *wait*, and *signal*. Following instruction, a student should: (1) understand how each synchronization primitive affects the state of operating-system-level resources, e.g., lock ownership, and the contents of condition and ready queues; and (2) be able to reason about higher-level thread interactions and synchronization behavior. This standard approach for teaching concurrency generally suffices for the first objective; however, many students find it difficult to achieve the second.

Clearly, this second objective represents a higher level of skill than the first (e.g., *application* vs. mere *comprehension* [4]). The ability to reason about the synchronization behaviors of threads requires the student to understand much more than just the operating-system-level semantics of synchronization primitives. She must also reason about thread interleavings and about how one thread's use of a synchronization primitive affects the scheduling of other threads, i.e., indirectly through the operating-system-level resources and states upon which these schedules are derived.

We intend to improve student learning with respect to the aforementioned objectives by evaluating and adapting the external representations used by experts in problem-solving tasks. That external representations can interact with internal representations to improve problem-solving is well-supported by work in distributed cognition [63]. Pancake states that purely textual representations may be inadequate to express complex concurrent programming situations [44]. Thus, visual representations are needed.

In a summary, the main goal of this research is to apply empirical methods to understand the difficulties novices commonly encounter in the comprehension of concurrent software, and to devise and evaluate versions of modeling notations that better support the comprehension tasks that deal with concurrency and synchronization. Specifically, this research:

1. empirically investigates the challenges encountered by students and novice practitioners engaged in learning about concurrency and synchronization;

2. empirically evaluates usability of existing notations, tools, and methods intended to address those challenges;

3. proposes, develops and empirically evaluates new notations and tools;

4. generates a body of reproducible, empirical knowledge that informs improvements in teaching proper use of concurrency and synchronization.

The remainder of the dissertation is structured as follows. By way of background (Chapter 2), we briefly survey the related work in empirical evaluation of the challenges dealing with concurrency and synchronization, several prior attempts to solve the challenges, and various graphical representations to support reasoning about concurrency and synchronization. Chapter 3 describes three empirical user studies, among which one study evaluated the usability of three existing graphical representations in dealing with the comprehension of concurrency and the other two studies aimed to investigate the fundamental problems that novices face when learning about concurrency and synchronization. Chapter 4 introduces our synchronization-adorned UML (saUML) sequence diagram notation and describes one subjective user study and three objective user studies that evaluated the usability of these diagrams when used as an aid in reasoning about concurrency and synchronization. Among the three objective studies, one compared the saUML sequence diagram notation with purely textual representations and the other two studies compared the saUML sequence diagram notation with the standard UML sequence diagram notation. Chapter 5 introduces how to use the UML 2.0 state machine diagram notation to model concurrent software systems and describes two empirical studies to evaluate its usability when used as an aid in reasoning about concurrency and synchronization—one compared it with purely textual representations and the other compared it with the standard UML sequence diagram notation. The

final chapter(Chapter 6) summarizes the contributions of our research to date and discusses future research plans.

Chapter 2

Background

2.1 Challenges dealing with concurrency

Several studies have looked at how students learn about concurrency and the kinds of problems that attend to this learning. Choi and Lewis [8] conducted a detailed study of student programs involving concurrency and synchronization and found that over 30% contained serious design flaws (e.g., data races, deadlocks, and inappropriate locking regimes) even though the programs produced correct output when tested against sample test cases.

A controlled experiment conducted by Kolikant [26] 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 shows 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 [7, 21]. For example, Higginbotham and Morelli [21] designed a concurrent programming interface based on the use of semaphores to correspond closely to some of the classical semaphore programming examples. Carr et al [7] designed a system, ThreadMentor, to visualize thread activities on-the-fly.

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. For example, when a thread invokes the *wait* primitive on a condition variable, the thread releases the mutex lock associated with that condition variable before suspending. Novices often fail to absorb this subtlety because the details are hidden in the implementation of *wait*, and the afore-mentioned tools have no way to expose this subtlety.

Kramer [27] observes that the ability to think abstractly is critical for a number of activities in computer science, including program analysis and comprehension of concurrent computations. He notes that only 30-35% of adults reach the formal operations stage of cognitive development at which such abstract thinking is supported. The prospects for success in these activities would seem dismal for the remaining 65-70%. He also advocates further studies to evaluate the effect of individual differences in abstract thinking capability on their performance.

However, studies of distributed cognition [63] provide some hope. Distributed cognition involves the interaction between *internal representations* and *external representations* in the performance of problem-solving tasks. Internal representations are described as existing "in the mind" as propositions, productions, schemas, mental images, connectionist networks, or other forms, while external representations exist "in the world" both as physical symbols (e.g., written symbols or the beads on an abacus) and the rules, constraints, or relations that pertain to those symbols (e.g., the relations among the symbols in a diagram or the physical constraints on the behavior of the abacus) [63]. Norman and Zhang showed that the choice of external representation affects the speed and accuracy of users engaged in such

problem-solving tasks. Thus, the choice of representation is important. The extent to which *good* choices of external representation can expand the population capable of successfully performing these tasks is an open question.

2.2 Representations dealing with concurrency

We believe that working with external representations can help novices to achieve a level of mastery that will permit them to better comprehend the dynamically evolving nature of concurrent programs, to more successfully engage in design, verification, and maintenance tasks for concurrent systems, and to retain the knowledge gained through use of these diagrams. In this section, we review a number of representations and modeling notations that deal with concurrency.

2.2.1 Representations based on formal methods

A Petri net [45], a mathematical modeling representation for the concurrent behavior of distributed systems, is well suited for describing and analyzing the synchronization and communication between concurrent processes. As a modeling language, it depicts the structure of a distributed system as a directed bipartite graph with annotations. Graphically, a Petri net comprises directed arcs that connect places (ellipses) and transitions (rectangles).

In practice, standard Petri nets have two major drawbacks: no data concept and no hierarchy concept [22]. Due to the lack of a data concept, each data manipulation has to be explicitly represented as places and transitions and hence the resulting model often becomes excessively large. On the other hand, the absence of hierarchy concepts makes it infeasible to build a big model that consists of individual sub-models.

However, those two drawbacks have been overcome by the development of the extensions to the standard Petri Net. CP-nets (Coloured Petri Nets) [22], one of the most well-known high-level Petri nets, is such an extension. CP-nets incorporate into standard Petri nets

the characteristics of a high-level programming language, which provides the primitives for defining data types and manipulating data values.

Another popular formal language that has adequate capabilities to model concurrency is FSP (Finite State Processes) [35]. A concurrent software system consists of multiple sequential processes, each of which can be described as a sequence of actions using a simple textual, algebraic FSP notation. For instance, if P and Q denote two FSP processes and x denotes an action, then "P=x→Q" defines that the process P engages in the action x and then behaves the same as the process Q. The interactions between processes are solely modeled through message-passing communications. Corresponding to each FSP definition, there is a graphical representation called LTS (Labeled Transition System), which displays and analyzes the behavior of the same models that are textually described by FSP.

Figure 2.1 presents a non-deterministic FSP, accompanied by its LTS, that models a coin in toss[1]. The "*mid*" symbol is a choice operator that represent a choice of multiple actions. If P, Q, and S denote three FSP processes, and x and y denote two actions, then "P=(x→Q|y→S)" indicates the process P may engage in either action x and behave as process Q or action y and behave as process S. When x and y denote the same action, then the choice is non-deterministic. The example shows that after the process COIN engages in the action **toss**, the subsequent action may be either heads or tails.

COIN=(toss→HEAD|toss→TAIL),
HEAD=(heads→COIN),
TAIL=(tails→COIN).

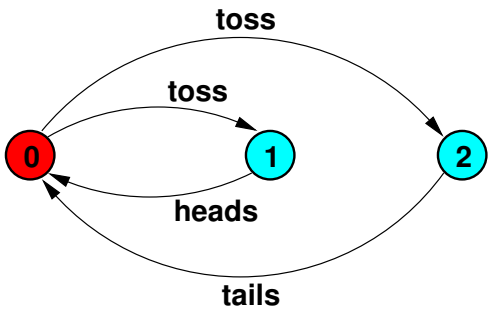


Figure 2.1: FSP and LTS that describe a process modeling a coin in toss.

These two formal methods and many others (e.g. the Z notation [51], VDM [1], etc.) share two major advantages. First of all, they are unambiguous. The rigor of their semantics

[1]This example is borrowed from [35].

often ensures the clarity of the specifications of a system design, which narrows the chances of fault injection during the system development. Secondly, the systems built upon formal specifications are open to correctness proof. The correctness of the system can be verified not only by human-directed mathematic proof, but also by model checkers, which exhaustively verify all the reachable states. However, both proof approaches require substantial human effort in the current state of practice. It may require much more effort to verify other important properties such as safety and liveness properties, as those properties are very problem-specific and sometimes require deep domain knowledge. This expectation is consistent with our experience [11]. Besides, writing precise specifications for even a small concurrent software system is not a trivial task to handle.

Despite their distinctive benefits and their extensive research development over the past decades, formal methods are still not applied widely. The biggest problem with formal methods is that they are very abstract, and thus difficult to learn and comprehend. Further, the mastery of such formal methods often needs a relatively high level of mathematical expertise and sophistication. Rather than using formal method representations as the instruments under research, we intend to investigate the representations that are more acceptable to novices and easier for novices to grasp. Meanwhile, we apply formal methods as effective utilities to facilitate our research projects. For example, we have applied FSP and LTSA to objectively assess the quality of the code snippets collected from one of our studies [11].

2.2.2 Unified Modeling Language

As opposed to the aforementioned representations, UML (The Unified Modeling Language) [47], the *de facto* industry standard for modeling object-oriented software systems, is an informal modeling language with ambiguous semantics [18]. We select the UML notations as our research target for their widespread acceptance in both academia and industry. Most entry-level computer science curricula have already adopted UML as the major object-oriented modeling language to teach and its learning curve is not as formidable as formal

methods. After a brief training, novice students can be good human subjects in empirical studies. This fact is particularly appealing, since our studies are mostly done in academic settings. Moreover, the educational benefit of the targeted external representations dealing with concurrency is also a focus of our research. From the collection of the UML notations, the *sequence diagram* notation and the *state machine diagram* notation have raised our special attention.

The UML sequence diagram notation

In this section, we briefly introduce the UML sequence diagram notation. More detailed specifications can be found in [47]. The UML sequence diagram is one of the two interaction views among the collection of UML diagrams. The other interaction view is the *communication diagram*[2]. A UML sequence diagram describes an execution scenario of a modeled system by revealing the interactions between objects in time order.

The UML 2.0 defines two kinds of objects—*active objects* and *passive objects*. An active object owns a thread of control while a passive object does not [47]. In UML sequence diagrams, each object is an individual participant in the interaction, represented graphically as a "lifeline". Notationally, the lifeline of an active object is a rectangle with doubled vertical lines on both its left and right sides and a vertical bar descending from its bottom edge, as seen on the left in Figure 2.2. The vertical bar represents the execution of a method call or an operation, referred to as an "execution specification" [3] in UML 2.0. The lifeline of a passive object is simply a rectangle with a dashed, vertical line descending from its bottom edge, as seen on the right in Figure 2.2. Execution specifications may be labeled on the lifeline of a passive object when its operations are called within the execution specifications of some active object. For both lifeline notations, the name and type of the object is placed within the rectangle in the form of **name:Type**. A lifeline may also be adorned with "object states" (also called "lifeline states") [47, pg 589], i.e, rounded rectangles containing the name of a

[2]formerly *collaboration diagrams* in UML 1.x.
[3]formerly *activations* in UML 1.x.

target state, to indicate the change of a "state" of the object, which is defined as a situation during the life of an object during which some invariant condition holds.

UML sequence diagrams represent the interactions between objects as sequences of message exchanges between lifelines over time. The call of an operation is often modeled by a pair of messages in UML sequence diagrams—a call message (a solid arrow line labeled with the message name and parameters, if any) and a return message (a dashed arrow line). Figure 2.2 presents a sample UML sequence diagram in which an active object *a* calls an operation *op* with a parameter *x* on a passive object *p*, during the execution of which the state of *p* changes from *state1* to *state2*. The call returns upon the completion of *op*.

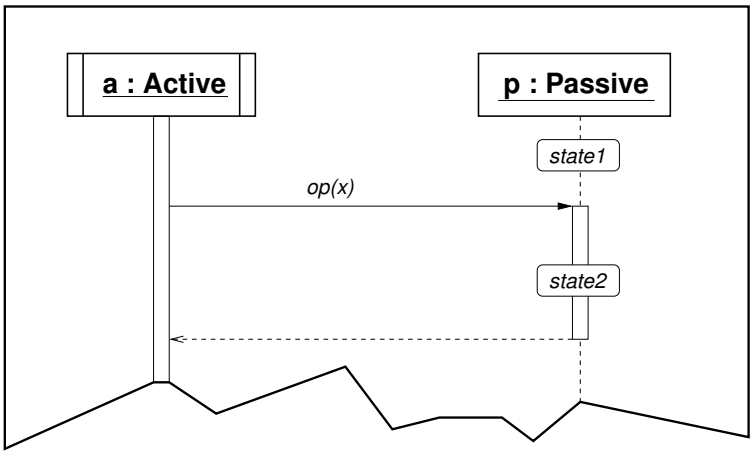


Figure 2.2: Sample UML sequence diagram.

We focused on UML sequence diagrams for several reasons. First, analysts often construct concrete scenarios of interaction when diagnosing faults in multi-threaded programs [17]. 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 [54]. 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. Moreover, while a given sequence diagram depicts only a single scenario, a small collection

of distinct sequence 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 UML sequence diagram also 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 threads as active objects and shared resources as passive objects, a sequence diagram visualizes the dynamics of synchronization, showing the time ordering of interactions between threads in a single program trace.

The UML state machine diagram notation

A single UML sequence diagram depicts only one execution trace of the modeled system; it does not cover all possible execution sequences. Another component of the UML diagram family, the UML state machine diagram, covers all potential life histories of an object and is better suited for the verification tasks of a program. In this section, we introduce the basic concepts of the UML state machine diagram notation.

In UML 2.0, a state machine diagram, a directed graph of states connected by transitions, models the lifetime behavior of an object [47]. Each state machine is an isolated entity that interacts with other state machines via events, which are occurrences that affect an object, such as the receipt of an operation invocation, a change in values, the passage of time, etc. The general notation for a state is a rounded rectangle enclosing the name of the state, such as the state **S1** in Figure 2.3. The notation differs for two special states—the "initial" state, shown as a filled circle as in Figure 2.3, which is the default starting state when the object is created and the optional "final" state, shown as a filled circle inside of a hollow circle as in Figure 2.3, which indicates the termination of the object[4]. The state notation may also include strings that label activities that the modeled object performs when it is in the state.

[4]The initial state and the final state have other semantics when enclosed in a "composite" state. More detailed descriptions can be found in [47].

A horizontal line separates the state name from these activity labels. For example, the state **S2** in Figure 2.3 contain a non-atomic activity **op** (e.g. counter++).

A "transition" is represented as a directed arc with an arrow head between a source state and a target state. A transition in general has an "event trigger", specified "guard conditions", and some "actions". The event trigger is the event whose occurrence makes the associated transition eligible to fire (e.g. buttonClicked). A guard condition, represented as a boolean expression enclosed in a pair of square brackets (e.g. [counter>0]), is a condition that must be satisfied to fire the associated transition. An action is a primitive, atomic computation that is executed when the associated transition fires (e.g. resetCounter()).

Figure 2.3 presents a sample state diagram that depicts a lifecyle of a simple object. Upon creation, the object enters the initial state. Then, being a triggerless transition, the transition from the initial state to state **S1** automatically fires and transforms the object to state **S1**. When the object receives an event trigger **e** and the associated guard condition **g** is satisfied, it executes the action **a** and enters the target state **S2**. While the object is in the state **S2**, it performs the activity **op**. The outgoing transition of the state **S2**, without being labeled with an explicit event trigger, is triggered by the completion of **op** and transforms the object into the final state.

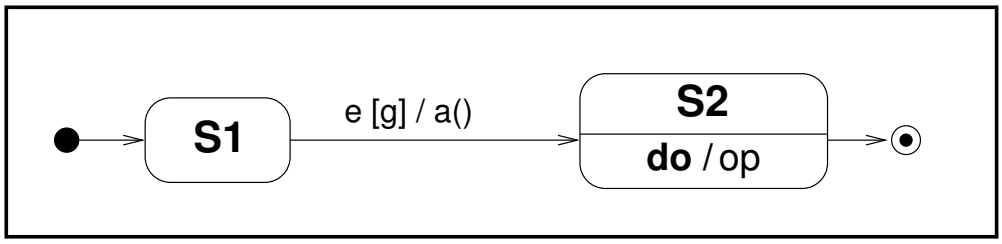


Figure 2.3: Sample UML state diagram.

Other related work of UML

Although UML does not have a systematic way to tackle the issues of concurrency, many concurrency features exist in its semantic specifications [42]. Stevens [52] surveyed the various

concurrency concerns illustrated in the UML specifications. There also have been a few attempts to model Java threads in UML notations. For example, Schader and Korthaus [48] examined the possibilities of modeling Java threads using UML and Mehner and Wagner [38] extended the UML to visualize the run-time mechanisms of the Java language constructs for synchronization. One group [32] specifically addresses concurrency and UML. This group has developed a tool, Jacot [33], to support the visualization of thread interactions in Java programs. The tool, however, is limited to the visualization of Java threads and, as with the other tools mentioned, provides only a high-level visualization of thread interactions.

A number of researchers have studied the usability of UML diagrams of one type or another. Swan, et al. studied a group of 40 senior-level computer science undergraduates, graduate students, and university staff with prior UML experience to compare sequence diagrams and collaboration diagrams [54]. They performed surveys of preferences and familiarity. They also evaluated performance and found that sequence diagrams were easier to use by those who were unfamiliar with either type of diagram, but that the diagrams were equally efficient for participants who were familiar with both types of diagrams. Kutar, et al. [29] also compared the impact of collaboration diagrams and sequence diagrams on comprehension, but found no significant difference.

Torchiano [57] compared the use of class diagrams alone to the use of class diagrams plus object diagrams as aids in answering questions about simple programs. In his study with 17 graduate software engineering students, he found that object diagrams provided benefits in some cases and were neutral in others.

Purchase, et al. [46] evaluated the effect of various aesthetic attributes (edge length, node distribution, and other layout characteristics) on viewers' comprehension of UML diagrams. Another group studied the effects of UML stereotypes (graphical icons) on program comprehension, as measured by performance and time to completion [30]. They found that use of the stereotypes both increased the rate of correct responses and decreased the time to completion.

Tilley and Huang [56] looked at the efficacy of UML diagrams in aiding program understanding. Subjects, UML experts, were given a series of UML diagrams and asked to answer questions about the depicted software system. They identified problems with layout, lack of support for representation of domain knowledge, and unclear specifications of syntax and semantics of some advanced UML features as limitations on support for program understanding.

Arisholm et al. [3] recently evaluated the impact of using UML modeling on the correctness and effort of certain software maintenance tasks. In particular, the authors were interested in evaluating the cost effectiveness of model-driven development with UML. They performed two experiments with students proficient in object-oriented programming and UML modeling. They found that the use of UML reduced the time required to make code changes. However, they found also that this savings is largely negated by the time required to modify the UML diagrams.

Known as an informal modeling language, the UML 2.0 specifications only define the abstract syntax of the UML diagrams without strict formal semantics. This is in particular the case for the UML state machine diagrams [34]. Fecher et al. listed twenty-nine newly detected unclarities in the semantics of the UML 2.0 state machine diagram notation [15]. In the absence of formal semantics of state machine diagrams in the UML 2.0 specifications, several researchers have attempted to define such a semantics [28, 34, 39, 58, 62]. For instance, Kuske provided a formalization of the UML state machine semantics based on structured graph transformation [28]. Yeung et al. [62] and Ng and Butler [39] formalized the UML state machine diagrams in terms of Communicating Sequential Processes (CSP) [5].

Chapter 3

Pilot study, Instructor interview and observational study

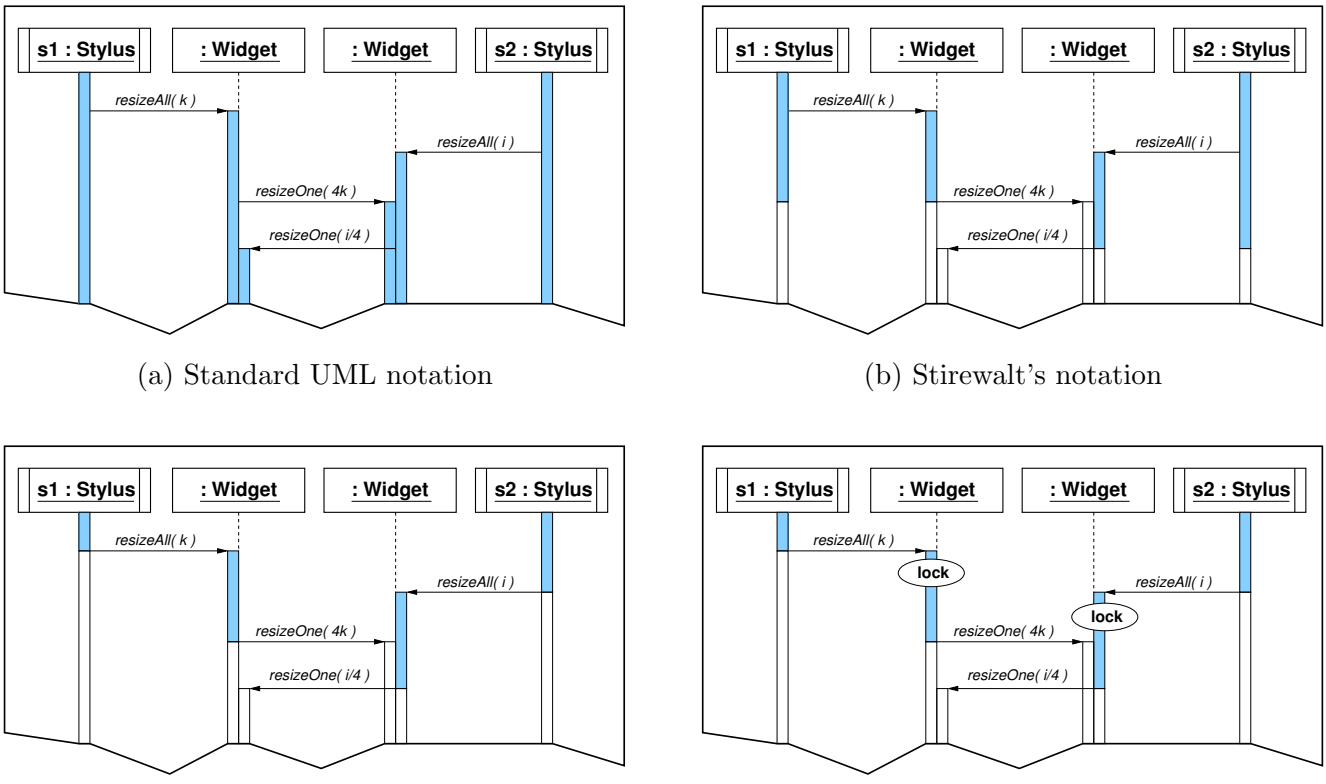
This research intends to optimize the efficiency of program comprehension in the context of concurrency and synchronization by designing and empirically evaluating external representations that are customized to address the factors that most programmers and analysts encounter. We began by conducting a comparative study to empirically evaluate the usability of the existing variations of UML sequence diagram notations in solving comprehension tasks involving multiple thread interactions. The results imply that a deliberately designed variant of the UML sequence diagram notation may provide better support towards the comprehension of concurrency concepts than the existing notations. We then focused on the investigation of the factors that complicate learning, with the idea that these same complexities would also complicate comprehension tasks. To do so, we conducted an instructor interview and an observational study [59] to understand the practical difficulties novices encounter in learning about concurrency. Such an investigation also guided us in the selection of the desirable properties of our new notation.

3.1 Comparative study of variations of UML sequence diagrams

This study was conducted at the University of Georgia from May 1st, 2006 to May 12th, 2006. The purpose was to evaluate the comparative ability of three variations of UML sequence diagram notations to help users evaluate concurrency concerns in those diagrams.

In detail, the three notations of interest were: Standard UML notation [47], Mehner's notation [38] and Stirewalt's notation, all seen in Figure 3.1. For all three variations, thick gray bars were used to indicate an activation of the object by some invoking thread. Mehner's

notation (Figure 3.1(c)) uses unshaded (white) execution specifications at any point when the activation is suspended, either because the thread that was executing within this activation is computing in some other activation (due to the invocation of a method in some other object) or because the thread is blocked waiting to acquire a lock on some object. Stirewalt's notation (Figure 3.1(b)) uses unshaded execution specifications only when the thread is blocked waiting to acquire a lock on some object. Meanwhile, we introduced an oval notation for object state, seen in Figure 3.1(d). 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: lock and unlock. These represented entry of the object into the locked and unlocked states respectively.



(a) Standard UML notation

(b) Stirewalt's notation

(c) Mehner's notation

(d) Mehner's notation adorned with object states

Figure 3.1: Variations of UML sequence diagram notation depicting a deadlock scenario in which two users concurrently manipulate a multi-user editor.

Twenty-four students, including eight undergraduate students and sixteen graduate students in the Computer Science department at the University of Georgia, were recruited to participate in this pilot study. We randomly divided all participants into six groups with each group consisting of four participants. Table 3.1 shows the grouping design.

Table 3.1: 3×2 Factorial Experimental Design - Number of subjects per group

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

The study was carried out using a questionnaire, which was produced in six versions. Each version included three sequence diagrams that depicted three different runs of a program:

- The first diagram depicted a non-blocking, non-deadlock interaction among a collection of objects.
- The second diagram depicted a blocking but non-deadlock interaction among the same collection of objects.
- The third diagram depicted a blocking and deadlock interaction among the same collection of objects.

In our experiment, we presented subjects with the three variations of sequence diagrams that model the behavior of a multi-user editor, which comprises a window that contained a rectangular widget. When one user used stylus $\mathbf{S}_1$ to resize the widget, a resize of the containing window was triggered. Likewise, when another user used stylus $\mathbf{S}_2$ to resize the window, a resize of the contained widget was triggered. Figure 3.1 presents examples of the three existing variations of UML sequence diagram depicting a program execution of the multi-user editor that runs into deadlock, which occurs when the two threads (represented by $\mathbf{S}_1$ and $\mathbf{S}_2$) are each holding a lock on a shared resource (widget or window)and waiting for another to release the lock. Note that Figure 3.1(d) presents an example in

which Mehner's notation is adorned with object states. The standard UML sequence diagram notation and Stirewalt's notation can also be adorned similarly with object states. All subjects anonymously answered fourteen questions after reading each diagram.

Our collected empirical results showed no significant differences in user performance across those notations except when considering only questions related to lock acquisition and release (p=0.024). Users from groups dealing with diagrams with ovals labeling the lock state were much more accurate in extracting information than others.

Lessons about experimental procedure and methodology learned from this pilot study include:

- The source code should not be presented in such studies. The source code was presented to all users, which made it easy for them to understand the interactions without trying to interpret the system through the notations.
- Questions with more gradations of difficulty should be used. In this pilot study, users scored high for easy questions but low for difficult questions. More intermediate-difficulty questions should be presented to better differentiate user performance.
- More complex scenarios should be presented. The presented software system was relatively simple. More complex scenarios can be used to better differentiate user performance.
- A training session should be introduced in the future studies. It might be that the subjects are most familiar with the standard UML diagram notation and thus performed somewhat better on it, which would indicate that we need to do a training session in future studies.

3.2 Instructor interview and observational study

We conducted an instructor interviews and an observational study to identify the common difficulties and misconceptions that students experience when learning about concurrency.

Three instructors of graduate and undergraduate-level courses that deal with concurrency concepts were individually interviewed. To stimulate discussion, the professors were given a list of key concurrency concepts and related example problems. They were asked to identify and describe the concepts students found difficult and how the instructors attempt to deal with these difficulties.

Next, to confirm the common problems pointed out by the instructors and to search for additional student difficulties, we conducted an unobtrusive observational study in an 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 about the instructor's presentation, the students' questions and the student responses to the instructor's questions. Common problems we identified include:

1. It is common for instructors to use ad-hoc sketches to describe concurrent program executions. However, students often find it difficult to record the details and explanations for later review of the reasoning behind the sketches.

2. The large space of potential execution sequences, each arising from a different thread interleaving, is difficult for a student to envision and comprehend.

3. Students often conflate the concepts critical region and scheduling policy to the point that they fail to consider execution sequences in which a thread executing within a critical region is interrupted due to context switching.

4. Students often become confused when answering what-if type questions in which the standard solution or primitive implementation is somewhat modified. That is, students have trouble reasoning about why the implementations of synchronization primitives lead to correct synchronization behavior.

5. Students often find it difficult to choose appropriate synchronization mechanisms and primitives to meet certain synchronization goals. This is consistent with the findings

by Shene and Carr [50], who claimed: “it is common for new MT (Multi-threaded) programmers to use very complex synchronization logic”.

Having identified the major difficulties that novices encounter when learning how to use concurrency and synchronization primitives, we then were ready to design and implement new representations that aim to help in addressing these difficulties.

Chapter 4

Design and evaluation of saUML

We designed the *synchronization-adorned UML* (saUML) sequence diagram notation to extend UML sequence diagrams with features to address the aforementioned difficulties commonly experienced by novices in learning about concurrency and synchronization [59, 61]. In the following sections, we introduce our saUML sequence diagram notation and describe four empirical studies that evaluated its usability.

4.1 saUML

Despite several attempts in earlier versions of UML sequence diagrams to model thread-level synchronization [48, 38], the current notation provides little support for modeling the various yet complex synchronization behaviors of concurrent threads [42, 52][1]. saUML was designed 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 [35], 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 an operation 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.[2] From now on, we use the term monitor to include such objects

[1] 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 [47].

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

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. In the following sections, we 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 [10].

4.1.1 Readers-writers: monitor solution

The readers-writers problem is a classic synchronization problem in which two distinct classes of threads exist, *reader* and *writer*. Multiple reader threads can enter the database simultaneously. However, no other writer thread, nor any reader thread, may be present in the database while a given writer thread is present. This policy is implemented using a protocol by which readers and writers request authorization prior to accessing the database by invoking *startRead* and *startWrite*, respectively. When readers or writers relinquish their access authorization, they need to call *endRead* or *endWrite*. Figure 4.1 presents C++–like pseudocode of a monitor solution to the readers-writers problem.

Figure 4.2 presents the implementations of the *wait*, *signal* and *broadcast* operations on condition variables. Note: (1) each condition variable maintains a *wait set* onto which threads may be placed to await resumption when some other thread invokes signal or broadcast for this condition variable; (2) *wait*, *signal* and *broadcast* are operations of class Object; thus the *release lock* statement releases the lock on the monitor object and does not affect the condition variable passed in as a parameter; (3) *Thread.currentThread()* returns the currently executing thread and *suspend()* causes the target thread to suspend execution until it is explicitly resumed by another thread.

4.1.2 saUML notational specifics

Our saUML sequence diagram notation extends the standard UML 2.0 sequence diagram notation [13] to explicitly represent phenomena related to synchronization. Figure 4.3 depicts

```
#include <ace/Thread_Mutex.h>
#include <ace/Condition_Thread_Mutex.h>
class Database{
  public:
    ...
    void startRead(){
        lock.acquire();
        while (numWriters > 0)
            okToRead.wait();
        ++numReaders;
        lock.release();
    }
    void endRead(){
        lock.acquire();
        --numReaders;
        if (numReaders==0)
            okToWrite.signal();
        lock.release();
    }
    void startWrite(){
        lock.acquire();
        while (numReaders > 0 || numWriters > 0)
            okToWrite.wait();
        ++numWriters;
        lock.release();
    }
    void endWrite(){
        lock.acquire();
        --numWriters;
        okToWrite.signal();
        okToRead.broadcast();
        lock.release();
    }
  private:
    ...
    unsigned int numReaders, numWriters;
    ACE_Thread_Mutex lock;
    ACE_Condition_Thread_Mutex okToRead, okToWrite;
}
```

Figure 4.1: A monitor-based solution to the readers-writers problem.

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

signal(ConditionVariable cond){
      choose t from wait set of cond;
          t.resume();
}

broadcast(ConditionVariable cond){
      for all t in wait set of cond;
          t.resume()
}
```

Figure 4.2: The pseudocode for *wait*, *signal* and *broadcast*.

a simple example. The diagram illustrates one possible scenario of interaction among three objects—r, d, and w. Our notation assumes that every class depicted in one of these diagrams must bear one of two stereotypes—thread or monitor[3].

We distinguish thread objects (e.g., r and w) using UML's double-bar convention for depicting active objects. Passive objects (e.g., d) are assumed to be monitors. Thus, the interaction depicted in Figure 4.3 involves a reader thread r, a writer thread w, and a monitor d.

Threads may be in one of three scheduling states: running, ready, or suspended. Our notation depicts the current scheduling state of a thread using colored execution specifications. By convention, the thread is running if its most deeply nested execution specification is shaded green, ready if this specification is shaded yellow, and suspended if it shaded red

[3]Not shown in this diagram, but would be apparent in the class diagram that defines classes Reader, Writer and Database.

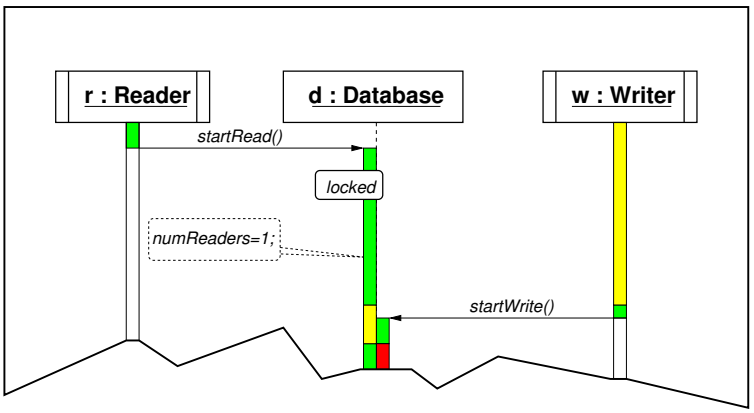


Figure 4.3: Sample saUML sequence diagram.

(varying intensities produce shades of gray that are distinguishable in monochrome displays or by color-blind users). Moreover, at any point in time, only the most deeply nested execution specification is shaded. In Figure 4.3, for instance, thread r is initially running, while thread w is initially ready.

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).

saUML depicts synchronization-relevant behavior using an idiomatic combination of UML features and nonstandard extensions to represent the synchronization state of a monitor and the scheduling state of a thread. Every monitor can be in one of two synchronization states—*locked* or *unlocked*—to represent whether a thread is currently executing within it. Changes to synchronization state are depicted using UML lifeline states. For instance, in Figure 4.3, 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, saUML uses comments to show the state changes of the counter variables *numReaders* and *numWriters*, which record the number of reader threads and writer threads, respectively, that are currently "in" (i.e., authorized to access) the database. Figure 4.3 shows that the reader thread increments the counter variable *numReaders* to one when it is executing within the monitor.

With these conventions in hand, a programmer would read the scenario depicted in Figure 4.3 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 *numReaders* 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.

Our extensions are most closely related to the sequence-diagram extensions proposed by Mehner and Wagner [38]. They color execution specifications to distinguish when an execution specification is active (dark) or suspended (white). These fine distinctions made by saUML address many of the knowledge gaps upon which novices often stumble. Mehner and Wagner also code *wait* and *signal* 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.

4.1.3 A walk-through of a program execution depicting by saUML

Figure 4.4 uses our saUML sequence diagram notation to present a complex interaction between a reader thread and a writer thread attempting to access a shared database. We

assume that only those two threads are running and that they are executing on a single physical processor. The numbers 1-11 shown to the left of the diagram are keyed to the following description:

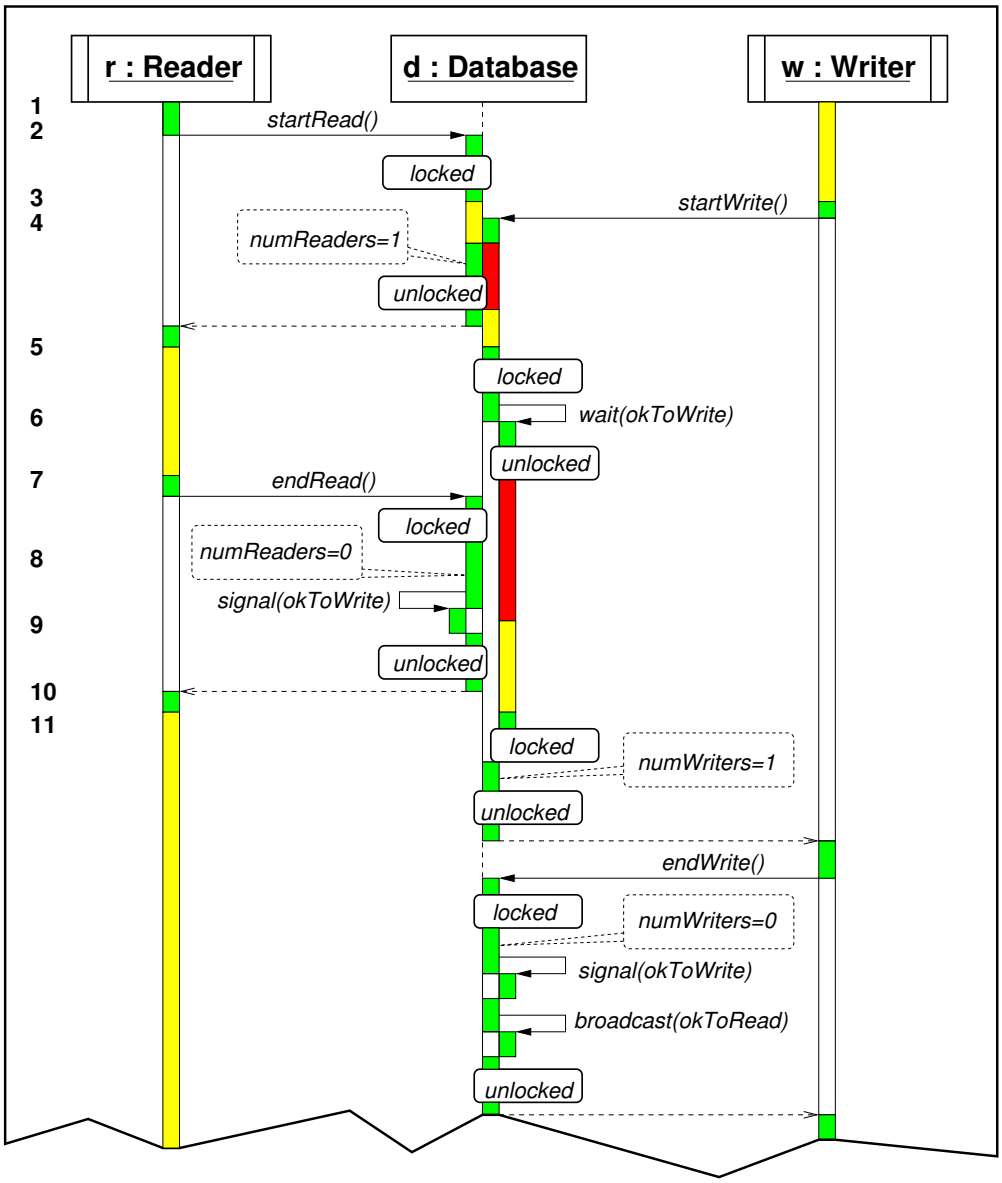


Figure 4.4: saUML diagram for the readers-writer example.

1. Initially, a reader thread r and a writer thread w are created "simultaneously". r is scheduled first; thus its execution specification is colored green (darkest gray). w

is ready (but not running) thus its execution specification is colored yellow (lightest gray).

2. r invokes *startRead()* on the monitor and is able to obtain the lock, as indicated by the appearance of the *locked* object state in the diagram.

3. A context switch occurs while r is executing *startRead*. w now runs (w's execution specification changes from yellow to green as r's execution specification changes from green to yellow).

4. w invokes *startWrite* and attempts to enter the monitor. However, the monitor lock is held by r and w suspends (execution specification changes from green to red). r resumes (execution specification changes from yellow to green). r sets numReaders to 1 (comment bubble), releases the lock (*unlocked* state) and returns from *startRead*. Notice that when r releases the lock, w's state changes from *suspended* to *ready* (red to yellow).

5. When another context switch occurs, w is able to run (green execution specification) and is able to obtain the monitor lock (*locked* state).

6. Because *numReaders* is non-zero, w invokes *wait(OKtoWrite)*. w then adds itself to the wait set of *OKtoWrite* (not depicted in diagram), releases the lock (*unlocked state*) and suspends itself (execution specification changes to red).

7. r now resumes its execution (green execution specification), invokes the monitor's *endRead* operation and acquires the lock (*locked* state).

8. r sets *numReaders* back to 0 (comment bubble) and invokes the monitor's *signal* operation on the condition variable *OKtoWrite*.

9. As a result of r's invocation of *signal(OKtoWrite)*, w, which is suspended and in the wait set of *OKtoWrite*, now resumes (execution specification changes from red to yellow).

10. r returns from the *signal* operation, releases the lock (*unlocked* state) and returns from its invocation of *endRead*.

11. Another context switch occurs, w is then scheduled (green execution specification). It obtains the monitor (*locked* state), and eventually completes its operation invocations.

We claim that our saUML diagram can help users to overcome several of the difficulties identified through our previous instructor interview and observation study. Using saUML diagrams rather than ad-hoc sketches during class should facilitate student note-taking and later review. Moreover, thread interleaving and context switching are depicted explicitly in saUML diagrams. Viewers can see from the diagrams that context switches can occur even when the thread is in a monitor or critical region.

saUML diagrams illustrate higher-level thread synchronization in a manner that is traceable to the use of low-level synchronization primitives. Thus, students can more easily answer a question such as, "What is the impact of the invocation of signal by the reader on the writer?" saUML diagram explicitly illustrates (in step 9) that the status of the writer changes from suspended to ready (the execution specification changes from red to yellow).

saUML can reveal the underlying dynamics of the synchronization primitives that are typically hidden from programmers. For instance, at step 5, when a context switch occurs, w enters the monitor (obtains the lock). What prevents w from writing while r is still reading? w sees that *numReaders* is 1, and suspends on *wait(OKtoWrite)*. Why does not this cause a deadlock? w releases the monitor lock (the *unlocked* state) before it suspends.

It is also possible to use saUML diagrams to ask why a thread is suspended. Can it not enter the monitor, or is it waiting on some condition variable? If it is waiting on a condition variable, the suspension will have begun during an activation of wait and should involve a transition from green to red that coincides with transition into the unlocked state.

4.2 Subjective study

We conducted a subjective user study of our diagram using five graduate students in the Computer Science Department at the University of Georgia who had recently been taught about semaphores and monitors. We first presented the code (Figure 4.1) and the saUML diagram (Figure 4.4) for the monitor solution to the readers-writers problem. We then walked the students through the event sequence described in the previous section and asked them to fill out a short survey (Appendix A). The first question surveyed familiarity with the standard UML 2.0 sequence- diagram notation. Of the participants, one had previously used the notation, one was familiar with but had not actually used it, and three were completely unfamiliar with it.

Questions 2-5 asked participants to use a rating scale from 1 (strongly disagree) to 5 (strongly agree) to evaluate how well the saUML diagram:

2. clarifies a thread's entering and exiting a monitor routine—average rating: 4.4;

3. 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;

4. illustrates the interactions between threads in a single program trace—average rating: 4.3;

5. facilitates my understanding of the inherent mechanisms of monitors—average rating: 4.0;

Question 6 asked participants if they could think of any other aspects of concurrency or synchronization behavior that this diagram might clarify or any ways in which it might aid in design, understanding or verification tasks. One participant suggested the diagrams could help distinguish when a thread is interrupted because it needs to synchronize with another thread vs. a "normal" context switch. Others suggested refinements to the saUML notation that included additional labels that record the values of condition variables, texture

to indicate execution specifications by a particular thread, and identification of the queue upon which a thread is blocked.

Question 7 asked participants to identify any aspects of concurrency or synchronization that this diagram might obfuscate or that might complicate design, verification, or understanding activities. One participant pointed out that the lengths of the sections of execution specifications could be misleading and perhaps should be scaled according to the amount of time that the depicted portion of the execution would actually consume. Another pointed out that it can be difficult to determine which execution specification is associated with which thread and that context switches could be difficult to detect. Perhaps these events could be more explicitly labeled.

Overall, the collected results are encouraging in that that all participants agreed or strongly agreed that this refinement of the sequence diagram is helpful.

4.3 More refined saUML

Based on the feedback collected from the subjective survey, we refined the saUML notation. We notice that when considering synchronizing multiple threads competing for access to some monitor objects, two major categories of synchronization states manifest: mutual exclusion synchronization states and condition synchronization states. While our initial saUML notation specifics employs two mutual exclusion synchronization states—*locked* and *unlocked*—to indicate whether the mutex lock used to guard access to monitor operations has been acquired or released, it does not deal with condition synchronization thoroughly. Recall that the early verions of saUML diagrams used comment bubbles to show the state changes of the counter variables. For instance, Figure 4.3 includes a comment bubble showing that *numReaders* has been set to one. However, such a representation appears to be informal and complicates the already-cluttered diagrams. It would be nice if we could use the existing, more formal UML features to represent condition synchronization states.

Moreover, condition synchronization states often associate values to problem-specific counter variables and conditions. When threads are synchronizing using condition variables, a condition change—a change of the counter variables—can be shown explicitly as a change of synchronization state on the lifeline of the monitor. Thus, both mutual exclusion synchronization states and condition synchronization states can be explicitly shown as UML lifeline states (i.e. object states) in parallel. Our more refined saUML notation allows for the depiction of zero or more problem-specific condition synchronization states, which are orthogonal to mutual exclusion synchronization states.

Figure 4.5 presents a more refined saUML sequence diagram that shows the same interaction as depicted in Figure 4.3. Notice that an orthogonal state appears in the lifeline of the monitor whenever a change to any of the mutual synchronization states and the condition synchronization states occurs. For example, the first orthogonal state corresponds to the event that r obtains the mutex lock on the monitor and the second orthogonal state corresponds to the event the r sets *numReaders* to one. Although the value of *numWriters* does not change in the presented diagram, its states are still included in the orthogonal states.

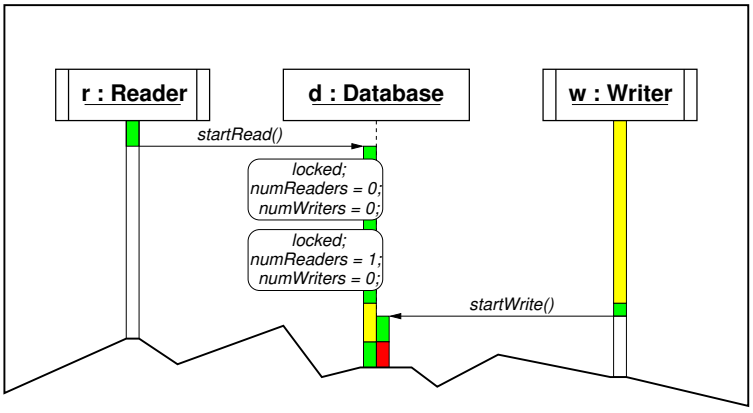


Figure 4.5: A more refined saUML sequence diagram.

4.4 Objective study: saUML vs. text-only

We designed the saUML diagram notation to address the obstacles to student learning uncovered by our instructor interview and observational study. The saUML extensions should help students in comprehension tasks involving the essentials of thread interleavings and context switches. The diagrams explicitly illustrate thread interactions and the effects of the execution of synchronization primitives by one thread on other threads. Consequently, they reveal more of the underlying dynamics of synchronization, information that is typically hidden from programmers. We expect that this external representation, designed to address some of the difficulties that users encounter in comprehending concurrent programs, will lead to improved performance on comprehension tasks.

We conducted an empirical study to examine the benefits of using our saUML sequence diagram notation in conjunction with source code in comprehension tasks for programs that employ concurrency. The data collected from the study showed a statistically significant benefit to the use of our notation.

4.4.1 Study design

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

Questions included in both the pre-test and the post-test were segregated into *knowledge-*, *comprehension-* and *application-level* questions. Bloom's taxonomy [4] categorizes the levels of abstraction of questions that commonly occur in educational settings. Knowledge-level

questions are the least abstract, and are characterized by recall of terminology of concepts. Comprehension is defined as "the ability to grasp the meaning of material," and may be demonstrated by translating material from one form to another, by explaining or summarizing material, or by predicting consequences or effects. Application-level questions require participants to use learned methods, concepts or theories in new situations or to solve problems. Obviously, application-level questions require a higher level of skills than both comprehesion- and knowledge-level questions. In the context of our study, knowledge- and comprehension-level questions evaluated participants' understanding of the concurrency and synchronization concepts presented in the lectures; application-level questions evaluated the ability to apply those concepts to solve model problems.

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

Two classic model synchronization problems—the shared bank account problem and the readers and writers problem [10] —were used in our study. The bank account problem, in which two or more threads concurrently access shared bank-account objects, is stated more formally as follows. Threads represent individual customers, some of whom may share bank accounts. Each customer may deposit or withdraw money from an account; however, to avoid a data race, only one customer may access any account at a time. A customer may execute a deposit at any time, provided that no other thread is currently accessing the account. A withdrawal requires both exclusive access and the condition that the balance be sufficient to permit withdrawal of the requested amount. Figure 4.6 depicts a Java-like pseudo-code

solution to this problem[4]. Notice that the *synchronized* Java directive indicates the implicit acquire and release of the monitor mutex lock upon entering and exiting the operations.

```
class BankAccount extends Object{

  private double balance = 0;
  private CondVar okToWithdraw = new CondVar;

  public synchronized void deposit(double amount) {
      balance = balance+amount;
      notifyAll(OKtoWithdraw);
  }

  public synchronized void withdraw(double amount) {
      while (amount > balance)
          wait(okToWithdraw);
      balance = balance-amount;
  }
}
```

Figure 4.6: Java-like pseudo-code solution to the bank account problem.

While the reader-writers problem has been introduced in previous sections, we now presents the Java version of its monitor solution (see Figure 4.7), as our study participants are mostly Java proficient.

4.4.2 First session

In the first 75 minute class session, the experimenter reviewed with the students the basic concepts and terminology in concurrency, including definitions and discussion of threads, context switches, race conditions, atomic operations, critical sections, deadlock, mutual exclusion, etc. A joint bank account example (with no synchronization mechanism) was used to illustrate race conditions. Condition variables and the monitor construct were then introduced, and a monitor solution to a simple rendezvous problem was presented. No diagrams were used.

[4]the *notify* operation and the *notifyAll* operation are equivalent to the *signal* operation and the *broadcast* operation, respectively.

```
class Database extends Object {

  private int numReaders = 0;
  private int numWriters = 0;
  private CondVar okToRead = new CondVar();
  private CondVar okToWrite = new CondVar();

  public synchronized void startRead() {
      while (numWriters > 0)
          wait(okToRead);
      numReaders++;
  }

  public synchronized void endRead() {
      numReaders--;
      if (numReaders > 0)
          notify(okToWrite);
  }

  public synchronized void startWrite() {
      while (numReaders > 0 || numWriters > 0)
          wait(okToWrite);
      numWriters++;
  }

  public synchronized void endWrite() {
      numWriters--;
      notify(okToWrite);
      notifyAll(okToRead);
  }
}
```

Figure 4.7: Java-like pseudo-code solution to the readers-writers problem.

The pre-test was conducted at the end of the first class session. In the pre-test, we asked both comprehension-level questions about concurrency concepts and application-level questions that required students to apply those concepts in interpreting the interactions between two concurrent threads (*c1* and *c2*) in a single problem. The eight knowledge-level questions were presented to refresh students' memory about the basic concurrency concepts and to make sure that the participants had the fundamental knowledge to understand the terminology used in the tests. Of the participants, 83% scored 100% on these eight knowledge-level questions and 100% scored 75% or higher, indicating that students had sufficient grasp of the terminology and concepts of interest.

Seven application-level questions were presented and served as the basis for evaluating student comprehension of the essential synchronization primitives and the behavior of concurrent programs. Figure 4.8 presents a sample scenario description and question from the pre-test. The question refers to the monitor solution to the bank account solution seen in Figure 4.6. Please see Appendix B.1 for all the pre-test questions.

```
Assume c1 is running within the invocation of deposit(100) and c2 is in the
suspended state (it was suspended on the monitor lock).

Question: If c1 releases the monitor lock and leaves the monitor, then:
  a. c1 changes to ready and c2 changes to running;
  b. c1 changes to suspended and c2 changes to running;
  c. c1 remains running and c2 remains suspended;
  d. c1 remains running and c2 changes to ready;
  e. Deadlock occurs.
```

Figure 4.8: A sample pre-test scenario and question.

Based on their scores on these application-level questions, we divided the students evenly into two groups—a control group and a treatment (diagram) group—for the post-test. The division was designed to produce two groups with equal means and standard deviations of pre-test scores. However, only twelve students in the class attended both the pre-test and the post-test sessions. Due to drop-outs between the pre-test and post-test, the actual groups

did not have the same mean score on the pre-test. The treatment group received a mean of 3.833 out of 7 with a standard deviation of 0.983 on the pre-test while the control group received a mean score of 4.667 out of 7 with a standard deviation of 1.366. Our analysis methodology took this difference in variances between groups into account.

4.4.3 Second session

During the next phase of the experiment, the groups attended parallel lecture sessions in different classrooms. Each lecture covered the basic features of monitors and a monitor solution to the bank account problem. Moreover, the PowerPoint slides used to cover this material were exactly the same for both the control and the treatment group. Following this initial lecture, the treatment group was then introduced to our saUML notation and the diagram depicted in Figure 4.9, which presents a complex interaction between two customer threads— *c1* and *c2*—attempting to access a shared bank account with an initial balance of $0. We assume that only those two threads are running and that they are executing on a single physical processor. These students were then led to "walk through" the event sequence conveyed in Figure 4.9:

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

2. *c1* invokes the *deposit(100)* operation and is able to obtain the lock, as indicated by the change in *a*'s synchronization state to *locked*. The orthogonal condition *balance=0* is also depicted.

3. *c1* increases *a*'s account balance by $100, as indicated by the change in synchronization state with the condition *balance=100* (monitor remains *locked*).

4. A context switch occurs. *c2* now runs (the shading of *c2*'s execution specification changes from yellow to green as that of *c1*'s changes from green to yellow). *c2* invokes

withdraw(150) and attempts to enter the monitor. However, the monitor lock is held by *c1*, so *c2* suspends (the shading of its execution specification changes to red).

5. *c1* then resumes (shading changes from yellow to green) and calls *notifyAll(okToWithdraw)*. Because no thread is suspended on the condition variable *okToWithdraw*, this operation does not trigger any changes in scheduling state but releases the lock (*unlocked* state) and returns from *deposit(100)*. Notice, at the point at which *c1* releases the lock, *c2*'s scheduling state changes from *suspended* to *ready* (red to yellow).

6. When another context switch occurs, *c2* is able to run (green shading) and is able to obtain the lock (*locked* state). Because the withdrawal amount ($150) exceeds the balance ($100), *wait(okToWithdraw)* is invoked. *c2* then adds itself to the wait set of *okToWithdraw*(not depicted in diagram), releases the lock (*unlocked state*) and suspends itself (shading changes to red).

7. *c1* now resumes execution (green shading). It invokes the deposit(100) operation, acquires the lock (locked state), and increases the *balance* by $100.

8. *c1* invokes *notifyAll(okToWithdraw)*. As a consequence, *c2* (suspended and in the wait set of *okToWithdraw*), now resumes to the *ready* scheduling state (shading changes from red to yellow).

9. *c1* returns from the *notifyAll* operation, releases the lock (*unlocked* state) and eventually returns from the invocation of *deposit(100)*.

10. A context switch occurs again. *c2* is then scheduled (green shading). It obtains the monitor lock (*locked* state), and because the current balance ($200) is now greater than the withdrawal amount ($150), *c2* eventually completes its transaction and finishes it execution, leaving the account balance at $50 (indicated in the orthogonal state).

The control group went through the same event sequence as the treatment group, as well as some additional examples to ensure that both groups spent the same amount of time

in this second teaching session; however the control group used only the program text (no diagrams).

Afterward, the two groups were brought together and received a brief introduction to the readers-writers problem and one of its monitor solutions (see Figure 4.7). This problem served as the basis for the majority of the questions and answers involved in the post-tests.

The post-test comprised seven scenarios of program execution traces, each followed by an application-level question. These questions were similar to those in the pre-test, except the pre-test questions addressed the simpler bank account problem rather than the more complex readers-writers problem. Figure 4.10 depicts a sample scenario and its associated question. In addition to these application-level questions, the post-test included three comprehension-level questions. All the post-test questions are presented in Appendix B.2.

All students then took the post-test. The control group received only textual materials while the treatment group received both the textual materials and saUML diagrams (Appendix B.3). The saUML diagram in Figure 4.11 corresponds to the scenario described in Figure 4.10.

4.4.4 Results and analysis

Table 4.1 summarizes the results of the study. The mean score of the control group dropped from 4.667 (67%) on the pre-test to 3.667 (52%) on the post-test, while the mean score of the treatment group rose from 3.833 (55%) to 4.833 (69%). We believe the drop in performance on the part of the control group was due to the use of a more complex problem (readers-writers) on the post-test than what was used on the pre-test. Structurally, the pre-test and post-tests were equivalent, as each contained only seven application-level questions with scenario descriptions of comparable length. To measure the effect that our notation had on the subjects' ability to perform problem-solving tasks, we compared the changes in scores from pre-test to post-test (post-test scores minus pre-test scores). By applying a two-tailed heteroscedastic (does not assume equal variance) t-test to the improvement matrix, we obtained

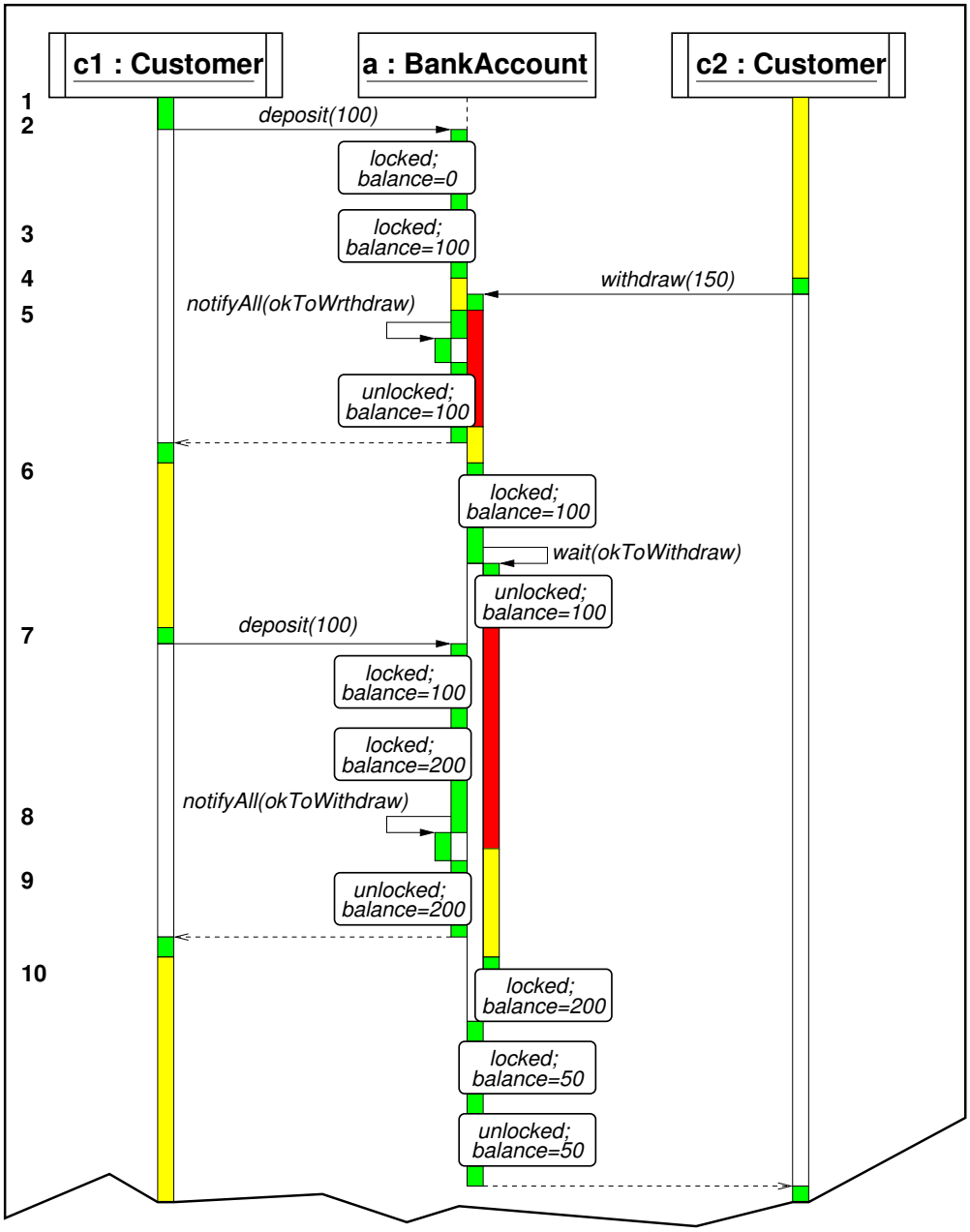


Figure 4.9: saUML diagram for the shared bank account example.

```
Assume the reader thread r is in the running state and the writer thread w
is in the suspended state (previously suspended on wait(okToWrite)). r
invokes endRead() and enters the monitor. It sets numReaders to 0 and issues
notify(okToWrite). Assume only the two threads are running on the processor.

Question: As a result:
  a. r changes to ready; w changes to running.
  b. r changes to suspended; w changes to running.
  c. r remains running; w remains suspended.
  d. r remains running; w changes to ready.
  e. Deadlock occurs.
```

Figure 4.10: A sample post-test scenario and question.

a p-value of 0.027. This result indicates a statistically significant difference between the two groups, with the treatment group outperforming the control group.

Table 4.1: Statistical results - Mean/Standard Deviation

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

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

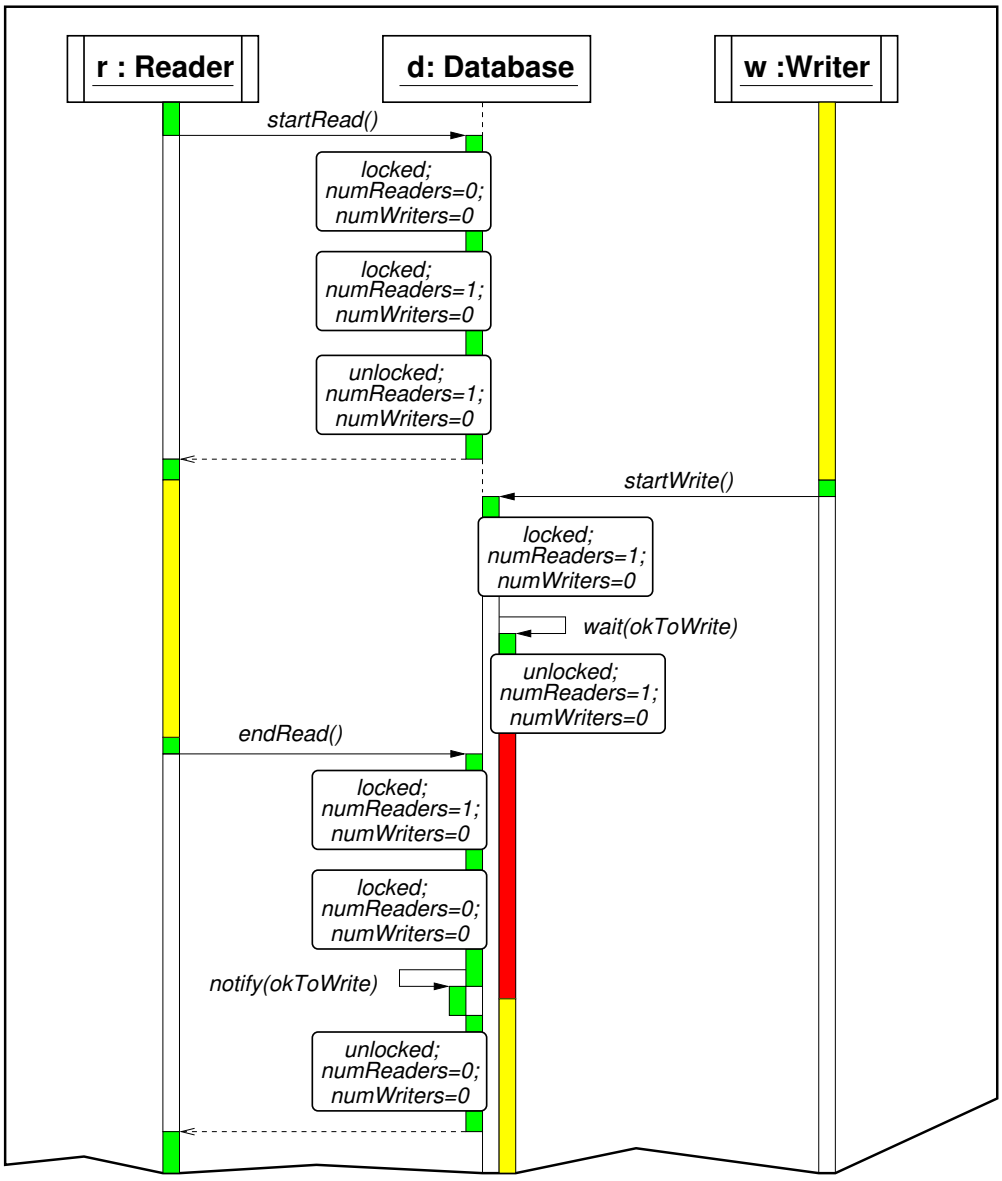


Figure 4.11: Sample saUML diagram appeared in the post-test.

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

4.5 Two Objective studies: saUML vs. standard UML

In the prior section, we reported how saUML sequence diagrams aid novice programmers in understanding tasks when compared with purely textual representations [61]. Whereas the positive effects of diagrammatic over purely textual representations are well documented [44], 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. We intends to investigate this question by comparing saUML with a simpler notation—standard UML 2.0 sequence diagrams.

To understand whether the benefits we measured in the previous study were due to saUML's specific extensions or merely to the use of a diagrammatic notation to accompany the text, 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 junior-level computer science students. For each study, students were partitioned into two equivalent groups as measured by scores on a pre-test. 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 control group referred to the code and to standard UML sequence diagrams depicting the same phenomena.

In the first study, the treatment group was more successful than the control group at answering questions regarding programs with simple synchronization. However, this improvement did not rise to the level of statistical significance. In contrast, the second study demon-

strated a statistically significant improvement of the treatment group at answering questions regarding programs with condition synchronization ($p < 0.05$). These results suggest that the saUML extensions play a role in improving performance in programs that employ condition synchronization. A benefit for programs with simple synchronization may or may not exist. Our study of the simple synchronization case involved 24 subjects. However, an *a posteriori* power analysis using the effect size and sample variance observed in this study indicates the need for a larger study, of 60-70 participants.

4.5.1 Study Design

We conducted two experiments to compare the effectiveness of saUML diagrams with that of standard UML sequence diagrams, when used as aids to the comprehension of concurrent systems. Effectiveness was measured in terms of the number of correct responses to questions about scenarios of concurrent program execution.

Both experiments employed a between-subjects, pre-test/post-test study design to compare the ability of novices to understand and answer questions about computer programs involving concurrency and synchronization using either (1) the traditional UML sequence diagram notation (control group) or (2) the saUML sequence diagram notation (treatment group). Participants were students in an undergraduate software design class at Michigan State University. 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 with more complex problems, involving condition synchronization.

Both experiments involved two eighty-minute sessions. Each session began with a lecture and ended with a test.

4.5.2 Experiment I

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. 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.

Session i

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 [59].

Figure 4.12 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` operation overlap in time. The experimenter would display such a

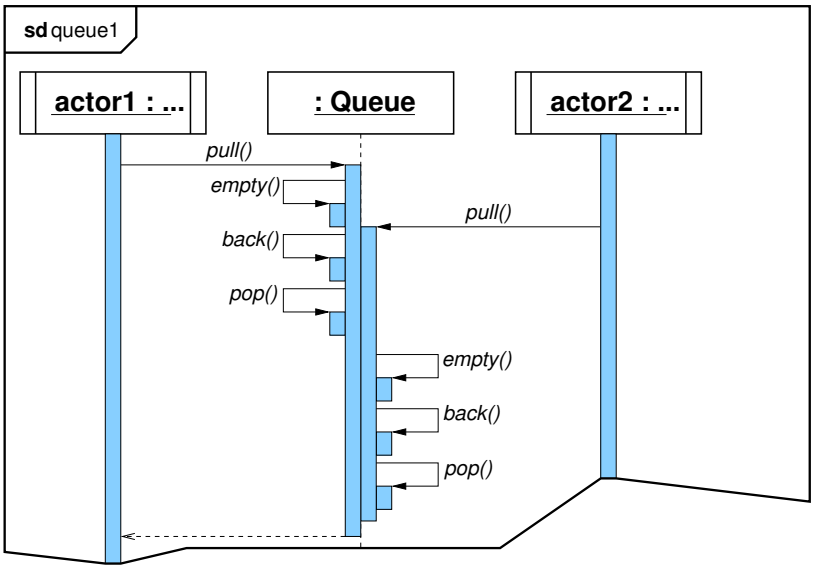


Figure 4.12: One sample scenario used in probing students' understanding of monitors.

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"[5]. Figure 4.13 depicts a scenario that is feasible in general but not feasible if the queue is implemented as a monitor.

Following this lecture, the experimenter then administered the pre-test. The pre-test contained nine application-level questions, which required participants to apply their knowledge of thread interleaving and synchronization in interpreting the interactions between two concurrent threads in a single program 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

[5]Students often fail to understand that a thread may invoke an operation 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.

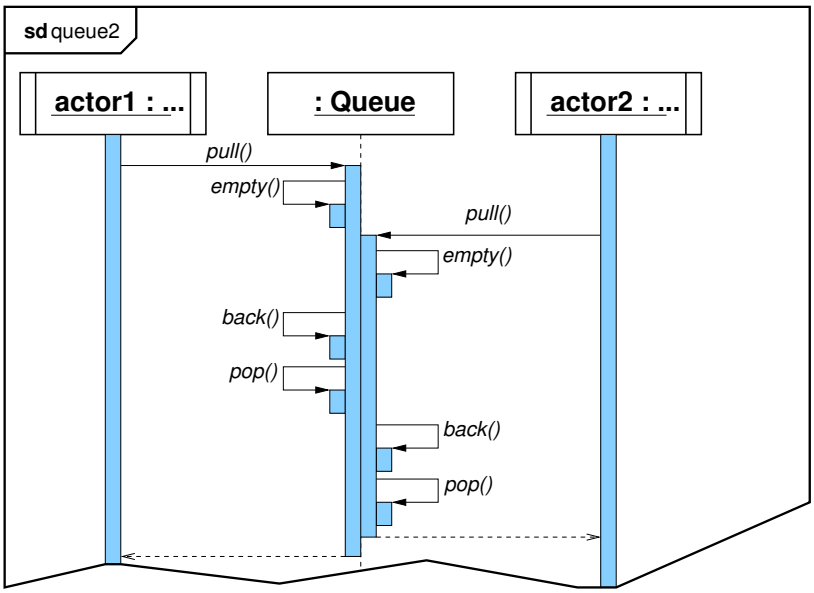


Figure 4.13: A second sample scenario used in probing students' understanding of monitors.

the interaction depicted. Students were asked to select the candidate that best described 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. Figure 4.14 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. For instance, this sample question targets the thread interactions depicted in Figure 4.12. Appendix C.1 presents all the pre-test questions.

Based on the participants' scores on the application-level questions on the pre-test, we divided them into a control group and a treatment group with equal means and standard deviations of score. Table 4.2 summarizes the results of participants' scores on the application-level questions of the pre-tests, normalized to the range 0.0 to 1.0. Some drop-outs occurred

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

    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.
    e. actor1 gets a copy of Object A; actor2 gets a copy of Object A;
       the Queue still contains Object A.
```

Figure 4.14: Sample question probing students' understanding of monitors.

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 4.2: Experiment I pre-test results, normalized

	Mean	Standard deviation
Treatment Group	0.694	0.207
Control Group	0.741	0.191

Session ii

The second session also 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 only in that the control group was presented with standard UML sequence diagrams, while the treatment group was presented with saUML 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 4.15: A monitor implementation of a shared bank account.

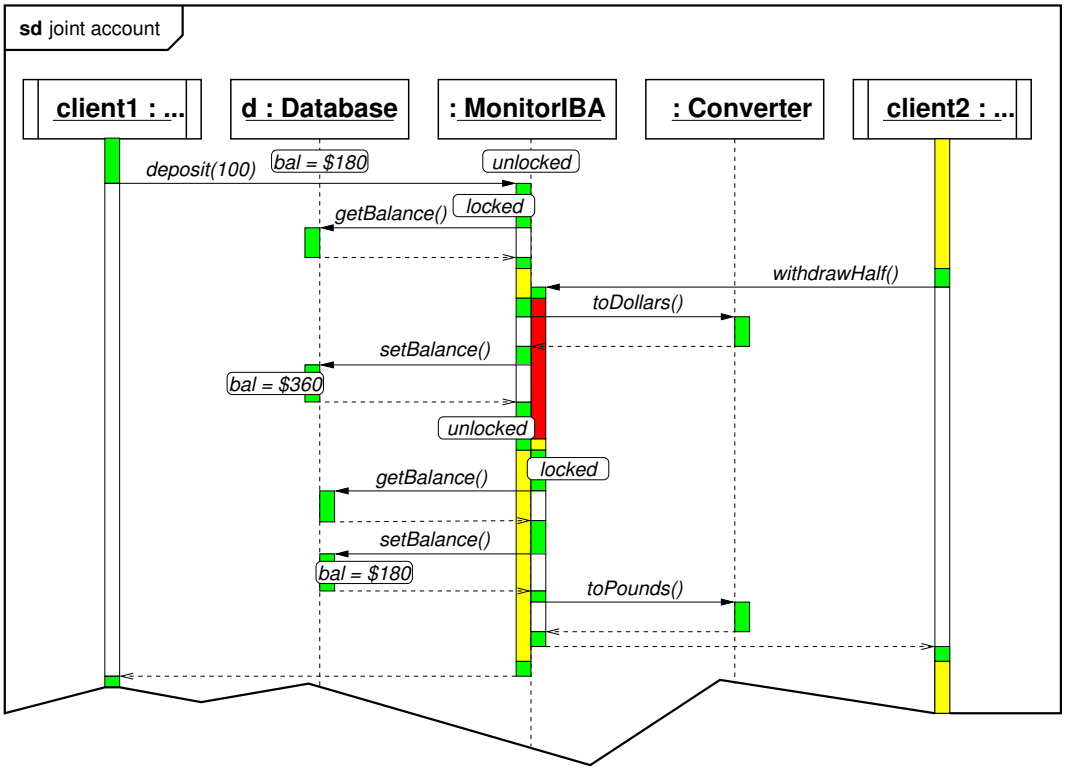


Figure 4.16: A saUML diagram used in the post-test of the first experiment.

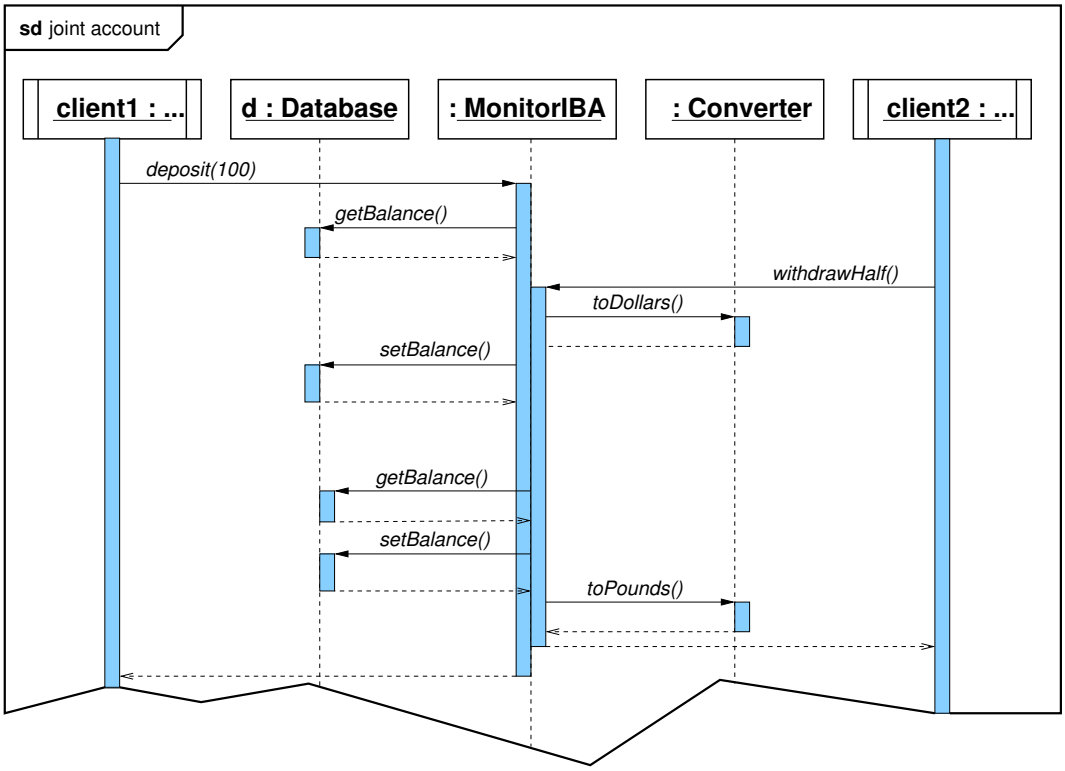


Figure 4.17: A standard UML sequence diagram used in the post-test of the first experiment.

```
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 4.18: Sample question from the post-test of the first experiment.

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 (Appendix C.2.1). In modeling the scenarios in the application-level questions, the treatment group was supplied with saUML sequence diagrams (Appendix C.2.2) while the control group was supplied with standard UML sequence diagrams (Appendix C.2.3). The nine application-level questions of the post-test were based on program execution scenarios involving a monitor implementation of a shared international bank account upon which two concurrent client threads invoked deposit and withdrawal operations.

Figure 4.15 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.

Figure 4.16 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)`[6], client2 suspends, waiting to enter the monitor. As indicated in the diagram, client2 remains suspended (red execution speci-

[6]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.

fication) 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 (Figure 4.17). Figure 4.18 presents a sample post-test question that targets the program execution scenario depicted in these diagrams.

Results and Analysis

Table 4.3: Experiment I post-test results, normalized

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

Table 4.3 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 application-level questions of the first experiment, 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 ($0.694 < 0.741$). However, this difference was not statistically significant, nor was the difference on the comprehension questions. Analysis was performed using both parametric (t-test with assumption on unequal variance) and non-parametric (Wilcoxon rank-sum) methods.

4.5.3 Experiment II

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. 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 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).

Session i

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 (Appendix C.3). Based on pre-test scores, 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 4.4 summarizes the means and standard deviations of the participants' scores on the eleven application-level questions of the pre-test, normalized to the range 0.0 to 1.0. The two groups had similar means and standard deviations of scores.

Table 4.4: Experiment II pre-test results, normalized

	Mean	Standard deviation
Treatment Group	0.672	0.202
Control Group	0.650	0.207

Session ii

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-level questions and eleven application-level questions (Appendix C.4.1). The application-level questions were based on execution scenarios involving a solution to the readers-writers problem [31], which involves condition synchronization and is more complex than the international bank account problem of the first study. In modeling these scenarios, the treatment group was supplied with saUML diagrams (Appendix C.4.2) and the control group with standard UML sequence diagrams (Appendix C.4.3).

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 *numReaders* and *numWriters*, and two condition-variable objects *okToRead* and *okToWrite*. These entities are declared as private data members of class `Database`.

The post-test questions refer to scenarios involving either two readers, one reader and one writer, or two writers. As an example, Figure 4.19 depicts the UML and saUML diagrams representing a scenario in which a writer invokes *startWrite* after a reader has been granted

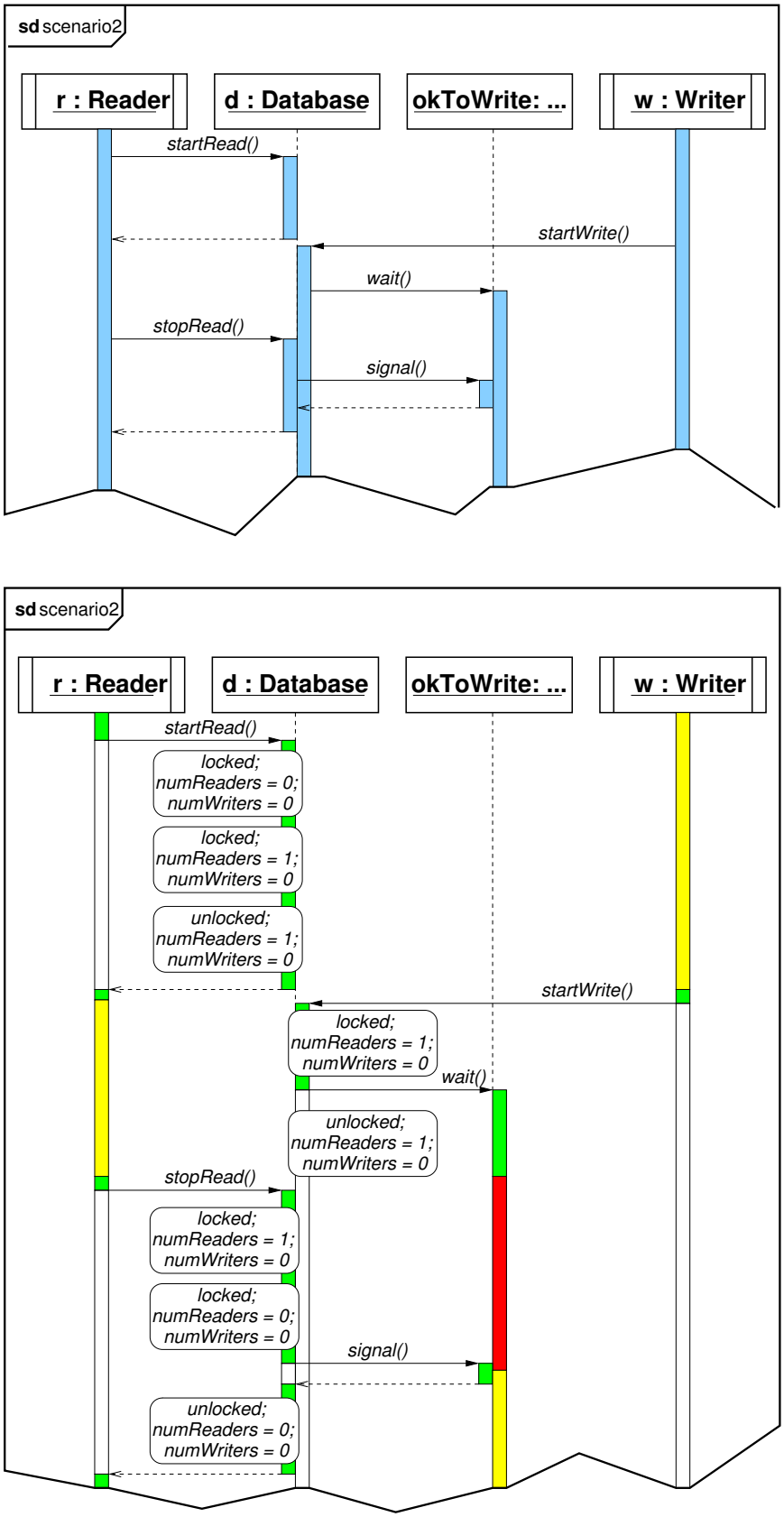


Figure 4.19: Sample post-test scenario depicted by UML and saUML renderings.

read access to the database. Immediately following the invocation of *startRead*, the counter variable *numReaders* 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*.

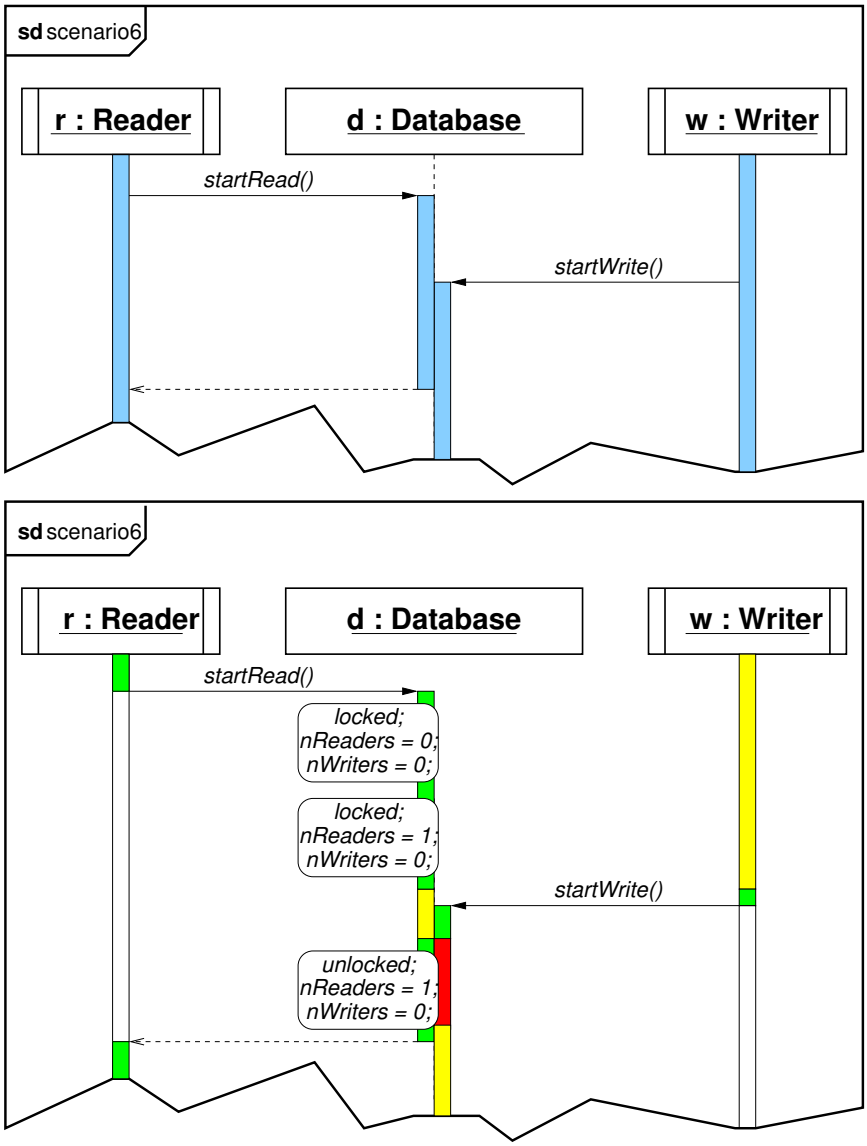


Figure 4.20: Another sample post-test scenario depicted by UML and saUML renderings.

Figure 4.20 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.

Results and Analysis

Table 4.5 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. No significant difference was found for the comprehension-level questions. 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 4.5: Experiment II post-test results, normalized

	Application-level		Comprehension-level	
	Mean	Standard deviation	Mean	Standard deviation
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 Figure 4.19, 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 Figure 4.20 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).

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.

Chapter 5

Evaluation of UML state machine diagrams as aids in the comprehension of concurrency concepts

In the object-oriented design idiom, a software system is seen as a collection of cooperating objects [2, 40, 55]. By applying the UML state machine diagram notation to model a running system, each object may be modeled by a distinct, isolated state machine, which communicates with other objects by detecting and responding to events [47]. Thus, each object appears to be acting as an active object that executes within its own thread of control. However, this interpretation is non-intuitive for passive objects, which do not initiate any control activity.

To address this issue, a commonly-used technique is to treat each state machine as a "sequential process" [35, 39, 62], which synchronizes with other processes by sending and receiving messages. Following this approach, we may appropriately differentiate the state machine representing a passive object from the state machine representing an active object by treating a passive object as a server process and an active object as a client process in the client-server style of interaction [53]. In the following section, we introduce how to follow this approach to model the synchronization behaviors of concurrent software systems.

5.1 client-server style state machines

In the client-server style of interaction, client processes and server processes communicate via a message-passing protocol—*rendezvous*, in which a client process sends requests in messages to a server process and blocks until the receiving server process replies. A server process

accepts a single client request from its associated entry, where client requests are queued up, and sends a reply message back to the sender upon completion of serving the request.

In UML 2.0, three actions **call**, **accept call**, and **reply**, described in [47, pg. 137-140], can be used to depict the interactions between client processes and server processes. However, these action types are not given any special visual syntax in UML. We adopt variants of the three primitive operations *call*, *accept* and *reply*, which are defined by Magee and Kramer [35, ch. 10] to express the rendezvous style of communication, in UML state machine diagrams as follows:

- /**call**($op(x)$)[1]

  An active object (i.e. a thread) invokes the operation *op*, which takes an argument x, of a passive object(i.e. a monitor). This is modeled by the client process sending the request to invoke *op* as a message to the entry of the receiving server process, then blocking until it receives a reply message.

- /**accept-call**($op(x)$)

  The server process receives a request from its entry. When the entry is empty, the server process blocks.

- /**reply**(op)

  A server process sends a reply message to the sender of *op* upon completion of the execution of *op*.

The correct use of the above notations requires several conventions. First, each named operation of a class is associated with an entry. However, no corresponding visual notation of an entry is presented in our diagrams. Second, we forbid event triggers when adorning a transition of the state machine diagram of a server object with **accept-call** actions. In this way, we avoid the scenario in which an event trigger fires a transition and then the server object executes an action that blocks indefinitely[2]. Third, we still allow guard conditions to

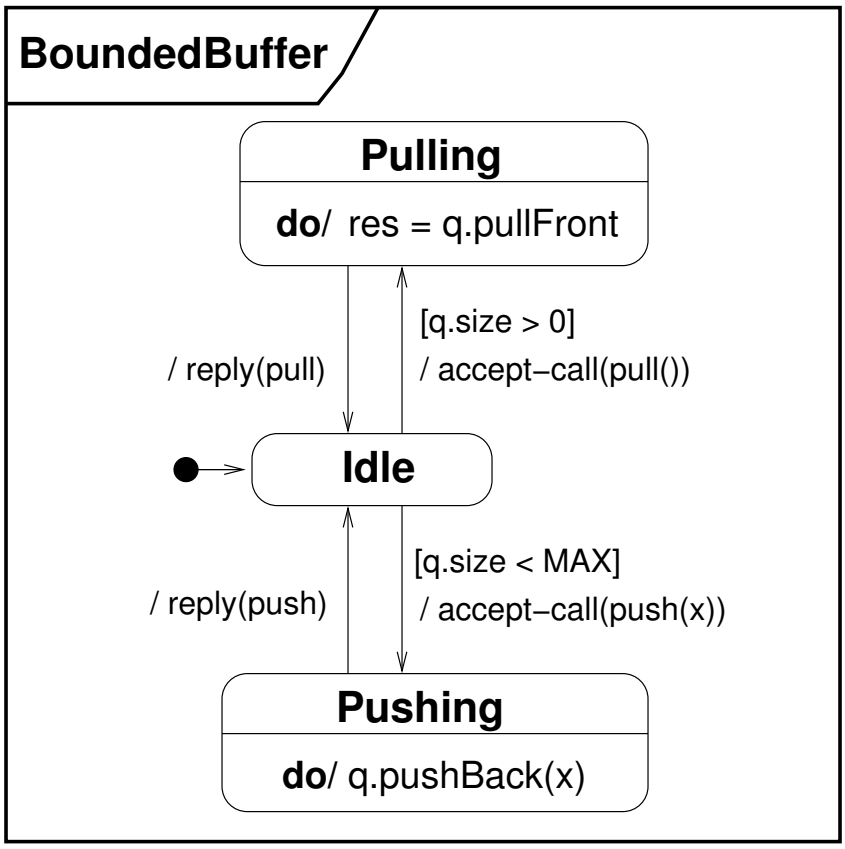
[1]The forward slash indicates that this expression is an action.

[2]Indefinite blocking is better modeled as an activity rather than as an action.

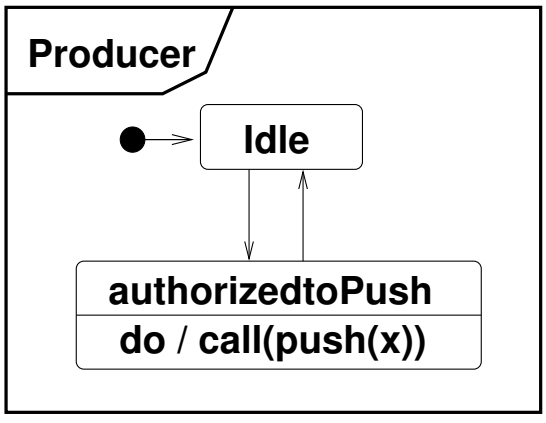
accompany the **accept-call** action, which may be enabled or disabled based on whether the corresponding guard condition is satisfied or not. Fourth, if any of the outgoing transitions of a state is labeled with an **accept-call** action, then all other outgoing transitions from the same state must be so labeled. Thus, the server object would not block waiting on a single **accept-call** action when other **accept-call** actions are ready to be executed.

In Figure 5.1, we present sample UML 2.0 state machine diagrams that model the synchronization behavior of a monitor-based implementation of the classic bounded-buffer problem, in which two processes—the producer and the consumer—share a limited-capacity buffer, using rendezvous semantics to specify both monitor and condition synchronization. Figure 5.1(b) and Figure 5.1(c) model two client processes that execute the **call** actions while Figure 5.1(a) models the monitor server process that executes the **accept-call** actions and the **reply** actions. Here is a sample interaction between the producer and the bounded-buffer. The producer calling the *push* operation of *BoundedBuffer* is modeled by it sending the *push* request to the bounded-buffer. After sending out the request, the producer then blocks until the bounded-buffer replies. Client requests are queued up at the entry of the bounded-buffer. The bounded-buffer accepts one request at time. When the bounded-buffer accepts the *push* request, and if the buffer is not full, it transforms to the **Pushing** state. Upon completion of serving the *push* request, the bounded-buffer sends a reply message to the producer, returns to the **Idle** state, and is ready to accept another request from the entry.

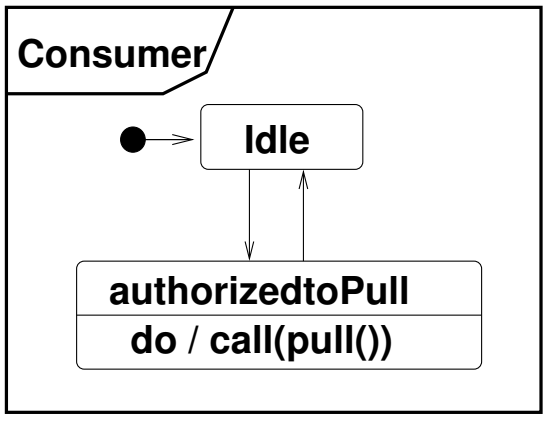
Figure 5.2 presents the C++ implementation code of two monitor operations—*push* and *pull*—of the *BoundedBuffer* class. It is worthwhile to notice that the traceability between this state model (Figure 5.1(a)) and the C++ code (Figure 5.2) is not obvious. That is, the state machine diagram does not reflect a subtle error in the synchronization logic, such as placing a *wait* on a condition variable in an *if* block rather than a *while* loop or forgetting a *signal*.



(a) BoundedBuffer



(b) Producer



(c) Consumer

Figure 5.1: UML state machine diagrams depicting a producer, a consumer and a bounded buffer, using rendezvous semantics to specify both monitor and condition synchronization.

```
void BoundedBuffer::push(int x)
{
  lock.acquire();
  while(q.size() == MAX) // MAX stores the capacity of the finite buffer;
    okToPush.wait();     // okToPush is a condition variable;
  q.pushBack(x);
  okToPull.signal();     // okToPull is a condition variable;
  lock.release();
}

int BoundedBuffer::pull()
{
  lock.acquire();
  while (q.size() == 0)
    okToPull.wait();
  int res = q.pullFront();
  okToPush.signal();
  lock.release();
  return res;
}
```

Figure 5.2: A monitor-based implementation of a bounded buffer.

A major goal of our research is to empirically evaluate the usability of the UML 2.0 state machine diagram notation when used as an aid in the comprehension of concurrency and synchronization. We recognize a mismatch between the UML state machine model of a concurrent software system and the underlying computational model employed by the same system. In the former, referred to as the message-passing model, state machines act as independent sequential processes which synchronize via message passing. In the latter, referred to as the thread-monitor model, the synchronization of concurrent threads is done by invoking operations on shared objects (the monitors). Such a mismatch is seemingly an obstacle to the usability of the UML 2.0 state machine diagram notation when it is used to model multi-threaded programs. In the following section, we describe an empirical study that we conducted to evaluate if UML state machines help users to understand concurrent executions and concurrency concepts or if their deviation from the thread-monitor model adversely impacts such an understanding.

5.2 Objective study: UML state machine modeling vs. non-modeling

In the fall of 2007, we conducted an empirical study to evaluate whether and to what extent UML state machine diagrams support novices in the comprehension, debugging, and implementation of code that employs concurrency. The study employed a between-subjects design with a pre-test and a post-test. The results suggest a beneficial trend towards the use of the UML state machine modeling.

As with our previous objective experiments on saUML, this experiment involved two eighty-minute class sessions, with each session starting with a lecture and ending with a test. Fifty-two undergraduate students from two Computer Science classes at Michigan State University participated in both sessions. Most of the students were juniors or seniors who had learned basic UML modeling and concurrency/synchronization concepts in either the current or previous semesters of study in computer science at Michigan State University.

5.2.1 Session 1

All participants attended the first class session together. To help the participants prepare for the study, the instructor spent the first fifty minutes in reviewing the fundamental concepts, terminologies and representations of concurrency and synchronization, UML 2.0 state machine models, and the rendezvous style message-passing models.

Participants spent the final thirty minutes taking the pre-test. The pre-test (Appendix D.1), which consisted of seventeen multiple-choice questions, was designed to investigate subjects' knowledge of concurrency and UML modeling. The seventeen multiple-choice questions consisted of thirteen comprehension-level questions and four application-level questions. The first nine comprehension-level questions covered basic concurrency and synchronization concepts such as the monitor construct, race condition, critical section, condition variable, deadlock, etc. The four comprehension-level questions concerned UML state machine diagrams and rendezvous style message-passing communication. The pre-test also included Figure 5.1(a), which served as a basis for four application-level questions targeting the state machine model of the bounded-buffer monitor object.

Based on participants' scores on the pre-test, we divided them into a treatment group and a control group so that the scores of the two groups conformed to approximately the same distribution. Table 5.1 summarizes the key statistical results of the two groups' scores on the pre-test, normalized to the range 0.0 to 1.0.

Table 5.1: State modeling vs. non-modeling: pre-test results, normalized

	Mean	Standard deviation
Treatment Group	0.593	0.144
Control Group	0.599	0.154

5.2.2 Session II

The two groups attended the second session together. The lecture component of the session introduced the use of UML state machine diagrams to model thread synchronization. The modeling techniques of applying rendezvous style communication were demonstrated using two examples—the bounded-buffer example (code presented in Figure 5.2 and state machine model presented in Figure 5.1) and the shared bank account example (code presented in Figure 4.6). Figure 5.3 represents the UML state machine model of the *BankAccount* monitor object.

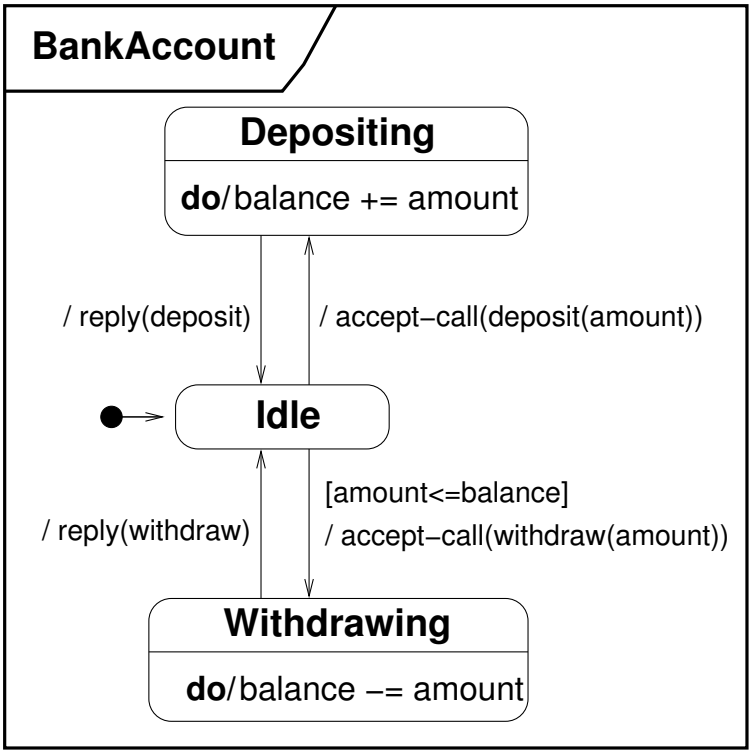


Figure 5.3: Abstract UML state machine model of the monitor *BankAccount*.

After the lecture, the two groups took the post-test. The post-test questions involved two complex synchronization problems—the readers-writers problem (Figure 4.1 on page 24) and the "match-maker" problem. Having discussed the readers-writers problem in Section 4.1.1, we now briefly describe the matchmaker problem.

The match-maker problem models the admission of (boy–girl) couples to a party. In detail, the matchmaking service maintains three counters (*numGirls*, *numBoys* and *numPairs*) and three monitor operations (*addGirl*, *addBoy*, and *pair*). Whenever a girl "arrives" (as a result of a client thread invoking *addGirl*), *numGirls* is incremented, after which the client may continue. Whenever a boy "arrives" (as a result of a client invoking *addBoy*), *numBoys* is incremented, after which the client may continue. Whenever both a girl and a boy are present, a match-maker client thread pairs them and decrements both *numGirls* and *numBoys*. Figure 5.4 presents a monitor-based solution to this problem.

Although not shown in the code, the condition variables (*goBoys* and *goGirls*) are both initialized with a reference to the monitor mutex (*lock*); thus, calls to *wait* on these entities will cause *lock* to be released. Figure 5.5 presents the UML state machine model of *MatchMaker*.

The post-test presented all participants with English descriptions of both the readers-writers problem and the match-maker problem, and prototype declarations of the monitor member functions—*startRead*, *endRead*, *startWrite*, *endWrite*, *addBoy*, *addGirl*, and *pair*. In addition to the textual materials, participants in the treatment group also received a UML state machine diagram that models the monitor *Database*. This diagram is presented in Figure 5.6.

Questions 1-3 in the post-test targeted the readers-writers problem. Question 1, referred to as the *true/false* question, required the participants to judge the feasibility of eight potential thread interaction executions of the monitor-based readers-writers implementation. Question 2, referred to as the error-correction question, presented erroneous implementations of the code for *startWrite* and *endWrite* (Figure 5.7) and required the participants to fix the errors. The three seeded errors are (1) missing *lock.release()* in *startWrite*; (2) missing *lock.acquire()* in *endWrite*; and (3) mistakenly using *okToRead.signal()* instead of *okToRead.broadcast()* in *endWrite*. Question 3, referred to as the *rw-coding* question, required the participants to fill the method bodies for *startRead* and *endRead*.

```
class MatchMaker {
  ACE Thread Mutex lock;
  ACE Condition Thread Mutex goGirls, goBoys;
  unsigned numGirls, numBoys, numPairs;
  ...
  void addBoy(){
    lock.acquire();
    ++numBoys;
    goGirls.signal();
    lock.release();
  }
  void addGirl(){
    lock.acquire();
    ++numGirls;
    goBoys.signal();
    lock.release();
  }
  void pair(){
    lock.acquire();
    while (numBoys <= numPairs || numGirls <= numPairs) {
      if (numBoys <= numPairs)
        goGirls.wait();
      if (numGirls <= numPairs)
        goBoys.wait();
    }
    ++numPairs;
    lock.release();
  }
};
```

Figure 5.4: Monitor-based implementation of *MatchMaker*.

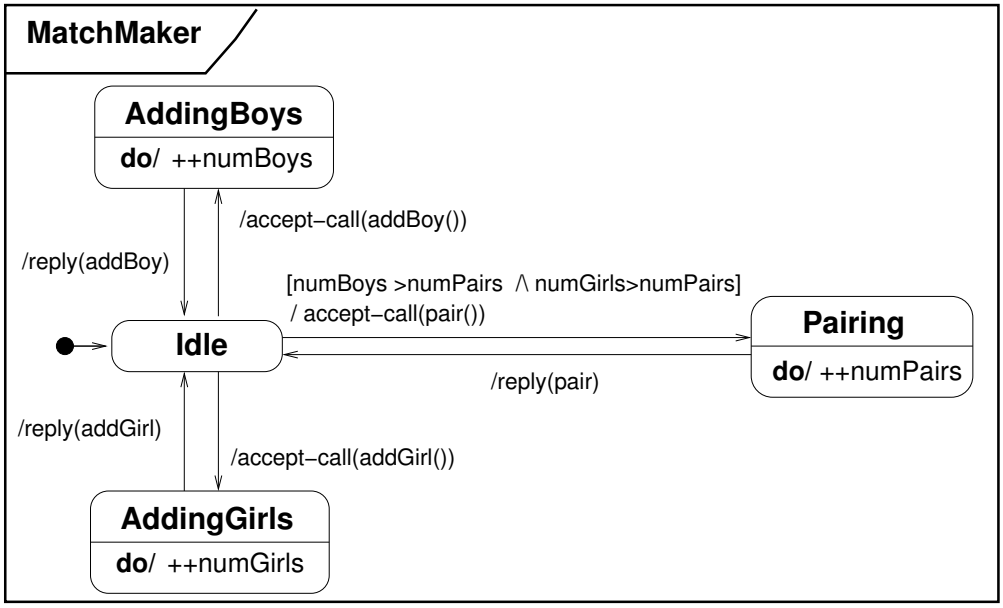


Figure 5.5: Abstract UML state machine model of the monitor *MatchMaker*.

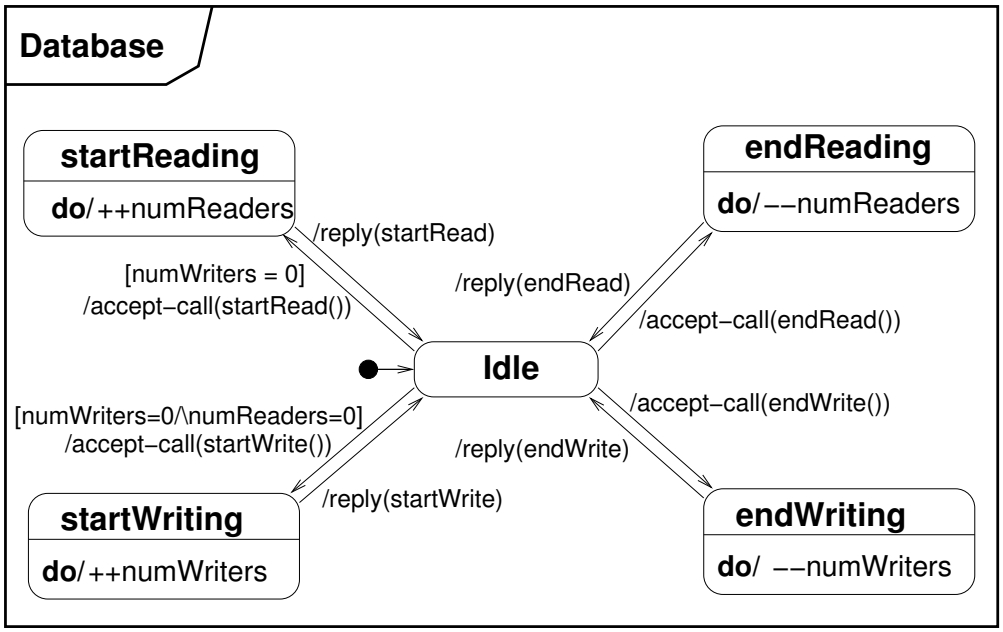


Figure 5.6: Abstract UML state machine model of the monitor *Database*.

```
...

void Database::startWrite(){
  lock.acquire();
  while (numReaders > 0 || numWriters > 0)
    okToWrite.wait();
  ++numWriters;
}

void Database::endWrite(){
  --numWriters_;
  okToWrite.signal();
  okToRead.signal();
  lock.release();
}
```

Figure 5.7: Erroneous implementations of the code for *startWrite* and *endWrite*.

Questions 4-5 targeted the match-maker problem. Participants in the treatment group were first asked to draw a UML 2.0 state diagram depicting the intended behavior of the shared match-maker object in question 4 and then asked to fill in the method bodies for the three operations (*addBoy*, *addGirl* and *pair*) in question 5, while those in the control group were first asked to fill in the method bodies in question 4 and then later asked to draw a UML 2.0 state machine diagram in question 5. We will refer to the state-diagram-drawing question as the *mm-drawing* question and the method-body-filling question as the *mm-coding* question in our future discussion. Appendix D.2 presents the test materials provided to the treatment group.

5.2.3 Results and analysis

We developed a set of correctness measures to grade participants' solutions to the two coding questions that required participants to fill in the method bodies. Table 5.2 summarizes those correctness measures.

Table 5.2: Correctness measures to participants' coding solutions

No.	Interpretation
1	Correct operations of *lock* acquiring and releasing
2	Correct operations on counters
3	Correct placement of *wait* on appropriate condition variables in *while* loops
4	Correct boolean conditions in *while* loops
5	Correct placement of *signal* or *broadcast* on appropriate condition variables

Measure 1 evaluates mutual exclusion synchronization criteria. In a monitor method body, *lock.acquire()* should be placed before any operation and *lock.release()* should be placed after any operation. Measure 2 evaluates whether the counter variables are updated correctly. Measure 3 evaluates the correct use of *while* loops around condition variable *wait* operations, in order to guarantee that the condition being waited for is still true after the thread returns from *wait*. Measure 4 evaluates whether correct boolean conditions were used to guard the *wait* operations within *while* loops. For instance, in the monitor operation *startWrite*, if the *while* loop assesses the boolean condition $numReaders > 0$ instead of $numReaders > 0 \; || \; numWriters > 0$, then multiple writer threads can access the share database simultaneously. Measure 5 evaluates correct use of condition variable *signal* and *broadcast* operations. For instance, if we use *okToRead.signal()* instead of *okToRead.broadcast()* in *endWrite*, then only one reader thread can be awakened. However, the desired synchronization behavior of the readers-writers problem is that all waiting reader threads shall be resumed if the last writer thread exits the database.

In assessing participants' state machine diagrams modeling the shared match-maker object, we defined the measures presented in Table 5.3. First, we expect a correct state machine diagram to include five states: the initial state, an idle state, and three states each indicating that the match-maker object is handling one of the three operation invocations—

addBoy, *addGirl* and *pair*. Next, if applicable, activities of updating counters should be present in the activity section of the state symbol. Last, a correct state machine diagram should have correct **accept-call** actions, **reply** actions and guard conditions adorned to the appropriate transitions.

Table 5.3: Measures to assess participants' state machine diagrams

No.	Interpretation
1	Correct number of states
2	Correct counter operations as activities
3	Correct **accept-call** actions depicted
4	Correct **reply** actions depicted
5	Correct guard conditions depicted

Table 5.4 summarizes the means and standard deviations of the two groups' scores on the selected post-test questions, normalized to the range 0.0 to 1.0. Overall, the treatment group performed better than the control group, as evidenced by higher mean scores on a per-question basis. However, with respect to the total scores on all five questions, the difference between the two groups is not statistically significant.

Table 5.4: Post-test results, normalized mean/standard deviation

	true/false	error-correction	rw-coding	mm-drawing	mm-coding
Treatment	0.665/0.186	0.640/0.229	0.520/0.227	0.520/0.330	0.432/0.249
Control	0.602/0.159	0.565/0.282	0.379/0.223	0.468/0.386	0.341/0.271

For the first two questions, the treatment group outperformed the control group, but the differences are not statistically significant. On one hand, the given state model of the shared database monitor object provided a concise overview of the underlying computational thread-monitor model, which helped participants in the treatment group to better reason about thread interactions. On the other hand, the poor traceability between the models and the implementation code did not assist much with the understanding of the operational mechanisms of low-level synchronization primitives such as *lock*, *wait*, *signal*. The first two

questions deal heavily with those synchronization primitives, which are not conveyed in the state models. For example, in the *error-correction* question, the seeded errors were related to incorrect operations on the monitor lock and condition variables. Such implementation details are implicit in the state models, but are not explicitly depicted. We believe this is the reason that not many statistically significant differences in quality between the two groups manifested for the first two questions.

A one-tailed, heteroscedastic t-test applied to the scores of participants' answers to the third question (the *rw-coding* question) shows a statistically significant benefit to the use of the UML state machine diagram ($p < 0.02$). By scrutinizing the collected answers, we found that while all participants of the treatment group included correct counter operations in their answers, only 22 out of 27 participants in the control group were able to do so. The difference is statistically significant ($p < 0.02$). We believe this difference can be attributed to the fact that the UML state machine diagram explicitly shows the counter operations as activities. This belief was confirmed by participants' answers to the *mm-coding* question. 21 out of 25 participants in the treatment group had correct counter operations, and only 12 out of 27 in the control group accomplished the same thing ($p < 0.01$). Further, the guard conditions depicted in the state models also helped participants to form correct boolean conditions of *while* loops. While 16 participants of the treatment group earned credits on this aspect, only 6 participants of the control group did so, resulting in another significant difference ($p < 0.01$).

Taking the first three questions regarding the readers-writers problem together, the treatment group significantly outperformed the control group ($p < 0.03$). However, when only considering the last two questions regarding the match-maker problem, no significant difference was found. This inconsistency is probably due to the fact that participants in the treatment group could rely on the provided correct state model for the reader-writer problem to answer the first three questions while, when facing the last two questions, they had to draw their own state models for the match-maker problem. The efficacy of their models was

seriously undermined by the reality that only 2 participants in the treatment group were able to draw correct state models for the match-maker problem.

In summary, correctly-developed UML state machine diagrams may provide significant benefits in supporting novices dealing with the comprehension, correction, and code generation tasks that employ concurrency.

5.2.4 Applying FSP models to assess the quality of concurrent programs

A potential threat to validity raised in the previous study is the objectivity of the grading scheme for measuring candidate solutions produced by subjects. We graded the collected code according to multiple quality measures. The scores were then used to perform statistical data analysis. Therefore, the soundness of our conclusions relies on the degree of objectivity of our quality measures, which is difficult to gauge.

Others have encountered the need to judge source code according to multiple quality measures in the context of an empirical study [13, 19]. Ideally, a distinct test case can be developed to assess each measure in isolation. However, code solutions produced during an empirical study may fail to compile or may take the form of code snippets rather than complete programs. Moreover, many non-functional qualities (e.g., extensibility) are not testable at all, and others are not reliably testable. Qualities related to correct use of synchronization primitives are good examples of the latter. Some researchers resort to subjective measures in these cases (e.g., [3, 24]); however, such measures may threaten validity and are difficult to weight relative to one another and to more objective measures.

We conducted a pilot study [11] to explore the use of formal modeling for determining the quality of candidate solutions collected from the UML state machine modeling study. To do so, we *faithfully* modeled five representative collected solutions to the "mm-coding" question and various synchronization-related properties in FSP [35] and used LTSA (Labeled Transition System Analyzer) [36] to analyze the solution models against the formalized properties.

A synchronization model is faithful to the code if (1) it is structured so that elements in the model correspond directly to code statements and structures and vice versa, and (2) it explicitly models low-level synchronization primitives (e.g., mutex locks and condition variables) and operations on shared data. By virtue of these properties, models are trivially traceable to code. Thus, synchronization flaws in the code will be preserved in the model rather than abstracted away, as may easily happen when constructing an ad hoc model of a program with a subtle flaw. This concern is especially relevant given the large number of such models that might need to be constructed.

Two questions we strove to answer through this pilot study are:

1. Given code that uses locks and condition synchronization to implement a shared resource, can we systematically generate a model that exhibits the same synchronization behavior as the code and that is compact enough to be feasibly analyzed?

2. Can we draw conclusions about the relative quality of the code through automated analysis of such models?

Faithful models

We formalized the five solutions in FSP models that bear two features that distinguish them from their more abstract counterparts—UML state machine models. First, they model operating-system-level resources, such as mutex locks, semaphores, and condition variables, as distinct behavioral entities. Our models borrow heavily on the conventions of Magee and Kramer [35, ch. 13], especially their model of condition variables. Second, model components are defined to correspond one-to- one to synchronization-relevant statements in the program and to compose according to composition of statements in the program. FSP models are nice in this regard: elementary program statements are modeled as sequential processes and composed using sequential composition, thereby mimicking the composition of statements in the program, and concurrency is modeled using parallel composition. These concepts are best illustrated by example.

```
||MATCH_MAKER_SHARED =
  ( LOCK_THREADS::MATCH_MAKER_LOCK ||
    MATCH_MAKER GO_GIRLS_CVAR ||
    MATCH_MAKER GO_BOYS_CVAR ||
    NUM_GIRLS_THREADS::MATCH_MAKER_NUM_GIRLS ||
    NUM_BOYS_THREADS::MATCH_MAKER_NUM_BOYS ||
    NUM_PAIRS_THREADS::MATCH_MAKER_NUM_PAIRS
  )
  ...
```

Figure 5.8: *MatchMaker* as a shared resource.

Figure 5.8 depicts the definition of an FSP process that models an instance of class *MatchMaker* as a shared resource using the idiom described in [35, sect. 3.1.3]. Under this idiom, operations in a shared resource manifest as actions, which are labeled by the set of threads that invoke the operation. `MATCH_MAKER_SHARED` is a composite process whose components correspond one-to-one with the data members of class *MatchMaker*, as depicted in Figure 5.4. The primitive process components (e.g., `MATCH_MAKER_LOCK`, `MATCH_MAKER_NUM_GIRLS`, etc.) are simply instances of reusable and pre-defined `MUTEX` and `COUNTER` processes, which are then labeled with the name of this resource (not shown). The *xxx*`_CVAR` processes model the condition variables *okGirls* and *okBoys*. These composite processes are instantiated in another part of the model from predefined processes using the idiom from [35, ch. 13].

By convention, the sets labeled *xxx*`_THREADS`, called *accessor thread sets*, contain the names of threads that access resource (or invoke operation) *xxx*. For instance, `LOCK_THREADS` names those threads that acquire or release the mutex lock; whereas `NUM_GIRLS_THREADS` names those threads that increment, decrement, or read the value of the counter variable *numGirls*. We use *accessor thread sets* so as to abstract the names of the actual threads out of the definition of the shared resource. This simplifies the assembly of analysis models, which will employ different configurations of threads. We also use accessor thread sets to

instantiate the condition variables (*goGirls* and *goBoys*). This instantiation (not shown) is more involved, but is nonetheless mechanical. We also elide a renaming of actions (ellipsis in the figure) needed to guarantee correct synchronization with client threads.

```
ACQUIRE_LOCK = (lock.acquire -> END).

INC_PAIRS = (numPairs.inc -> END).

PAIR_OP = ACQUIRE_LOCK; PAIR_OP_LOOP; INC_PAIRS; RELEASE_LOCK; END.

PAIR_OP_LOOP = (numBoys.read[nb:BRANGE] -> numPairs.read[np:PRANGE] ->
                  if (nb <= np)
                  then WHILE_BODY; PAIR_OP_LOOP
                  else . . .).

FIRST_IF = (numBoys.read[nb:BRANGE] -> numPairs.read[np:PRANGE] ->
                  if (nb <= np)
                  then WAIT_GO_BOYS; END
                  else END).

WAIT_GO_BOYS = (goBoys.wait -> lock.release ->
                  goBoys.endwait -> lock.acquire -> END).
```

Figure 5.9: Faithful model of the *pair* operation.

To model the operations of class *MatchMaker*, we first model each statement that directly accesses or modifies a synchronization object or counter variable as a sequential process[3]in FSP. We then use FSP's sequential composition and branching primitives to compose these primitive processes according to the control flow graph of a participant's submitted code.

Figure 5.9 depicts some of the FSP processes we defined to model the *pair* operation from Figure 5.4. The first three processes model statements that acquire the lock, release the lock, and increment *numPairs*, respectively. Process `PAIR_OP` is the sequential composition of subprocesses, each of which models one of the top-level statements of the method. The most interesting subprocess, `PAIR_OP_LOOP` models the while loop. It models checking the while condition using actions that read the counters *numBoys* and *numPairs*. Based on

[3]i.e., a terminating process—one that ultimately evolves into the primitive process.

the values read, it calculates the value of the first disjunct of the while condition. If true, control transfers into the body of the loop and then the process repeats. Otherwise, the second disjunct is checked (not shown in figure). Process `WHILE_BODY` (elided for brevity) is the sequential composition of two smaller processes—`FIRST_IF` (shown) and `SECOND_IF` (not shown). Process `FIRST_IF` models the first if statement inside the body of the loop. Process `WAIT_GO_BOYS` models invocation of the wait statement inside this first if block according to the idiom in [35, ch. 13].

These process definitions model the code to a high degree of fidelity. Process `PAIR_OP_LOOP` models reading the values of the counter variables in the order these reads would actually occur. Also, the process is structured to short-circuit evaluation of the second disjunct if the first disjunct is true. These process definitions could be automatically assembled from a sufficiently rich collection of primitive processes. Fortunately, in an empirical study, we usually know the primitive resources available and all operations *a priori*. In practice, these are often given as part of the problem statement.

Analysis models and properties

Prior to analysis, the FSP process created to model a shared resource must be composed with a process that models clients that actively invoke operations on the resource. We refer to the process modeling the shared resource as the *resource model*, the process modeling the clients as the *client model*, and to the parallel composition of the resource model and the client model as the *analysis model*. The Labeled Transition System Analyzer (LTSA) is used to check an analysis model for properties which may reveal the presence (or demonstrate the absence) of specific synchronization errors.

Of course, the properties satisfied by an analysis model depend not just on the resource model, but also on the client model. This fact is important, especially when most of the candidate solutions have synchronization errors, as it permits us to differentiate resource models that exhibit the same synchronization error. Consider, for example, the 5 client

models described in Figure 5.10 (bottom left and right), which are easily expressed in FSP as processes. These client models form an ordered set of progressively more concurrent (less restrictive) models, ranging from a strictly sequential client model (`SRou`) to a fully concurrent one (`FCon`). The more restrictive client models tend to mask certain concurrency errors. Moreover,if the parallel composition a given resource model with a given client model does not satisfy a given safety property, then the parallel composition of the given resource model with any less restrictive client model also does not satisfy the given safety property. Thus, for each safety property that expresses a desired quality of a solution, the ordering of the 5 client models induces an ordering on the 5 analysis models obtained using a given resource model. The analysis model in which a synchronization error first manifests provides a measure of the quality of the resource model (and thus of the associated candidate solution). The results of our pilot study illustrate this phenomenon more concretely.

LTSA automatically checks the analysis models for deadlocks. To assess other quality measures with LTSA, we defined 3 problem-specific properties. The first checks that a resource model conforms to the standard protocol for locking a shared resource—i.e., that a thread holds the lock when it invokes an operation on the data members of the shared resource and that it no longer holds the lock when it returns from an invocation of *addBoy*, *addGirl*, or *pair*. The second property checks that boys and girls are paired correctly—i.e., it tracks the number of times each counter has been incremented and checks that the number of increments of *numPairs* never exceeds the number of increments of *numBoys* or of *numGirls*. The third process checks for excessive signaling. While not a correctness requirement, this latter check provides insight into quality of a resource model. The definition of excessive signaling is both problem- and client model-specific. For our problem and a client model that calls each operations 3 times, the analysis model should generate a maximum of 6 signals (one for each invocation of *addBoy* or *addGirl*).

Deadlock					
Solution	SRou	CRou	TPMe	CPai	FCon
Correct	0	0	0	0	0
No.1	0	0	0	X	X
No.2	0	0	0	0	0
No.3	0	X	X	X	X
No.4	0	0	X	X	X
No.5	0	0	X	X	X

Lock protocol					
Solution	SRou	CRou	TPMe	CPai	FCon
Correct	0	0	0	0	0
No.1	0	0	0	–	–
No.2	0	0	0	0	0
No.3	0	–	–	–	–
No.4	0	0	–	–	–
No.5	0	0	–	–	–

Correct pairing					
Solution	SRou	CRou	TPMe	CPai	FCon
Correct	0	0	0	0	0
No.1	0	0	0	X	X
No.2	0	0	0	X	X
No.3	0	–	X	X	X
No.4	0	0	X	X	X
No.5	0	0	–	–	–

Excessive signaling					
Solution	SRou	CRou	TPMe	CPai	FCon
Correct	0	0	0	0	0
No.1	X	X	X	X	X
No.2	0	0	0	0	0
No.3	0	X	X	–	–
No.4	X	X	X	X	X
No.5	0	0	–	–	–

Client Models	
`SRou`	(Sequential Rounds) 1 thread calls `addBoy`, `addGirl`, & `pair` in sequence, 3 times
`CRou`	(Concurrent Rounds) 3 threads, each calls `addBoy`, `addGirl`, & `pair` in sequence, once
`TPMe`	(Thread Per Method) 1 thread per method, each calls its method 3 times

Client Models	
`CPai`	(Concurrent Pairs) 2 threads call `pair` once, 1 thread calls `addBoy` twice, & 1 thread calls `addGirl` twice
`FCon`	(Fully Concurrent) 2 threads call `addBoy` once, 2 threads call `addGirl` once, & 2 threads call `pair` once

Figure 5.10: Results of analyzing resource models

Preliminary findings

Figure 5.10 summarizes the results from our pilot study. We created resource models from the correct solution depicted in Figure 5.4 (indicated as `Correct`) and from 5 representative candidate solutions (indicated using participant numbers) submitted by participants the larger empirical study. We also created 5 client models (`SRou`, `CRou`, `TPMe`, `CPai`,and `FCon`). These represent different configurations of threads executing in client code, as described in the tables titled `ClientModels` (bottom left and right). The 6 resource models and 5 client models were composed, producing 30 analysis models. Each analysis model was checked for deadlock (top left), conformance to the lock protocol (top right), correct pairing of boys and girls (middle left), and excessive signaling (middle right). To show the results of safety checks, we mark: "0", if the model satisfies the property; "X", if the model may violate the property; and "–", if analysis is inconclusive.

We consider analyses inconclusive for two reasons, only one of which we anticipated. An analysis model composed from a faithful resource model can easily be too large for exhaustive analysis to be feasible. However, we expected that the solutions submitted by participants in our empirical study were simple enough that faithful models would also be analyzable, and indeed, for the most part, this was the case. We had to reduce the number of times each operation is invoked from 3 to 2 in the last 2 client models (`CPai` and `FCon`) in order not to exceed the Java stack size when using LTSA with these models. Fortunately, the properties checked in the pilot study did not depend on the precise number of operation invocations so long as each operation was invoked at least twice and the same number of times.

The second reason that analysis might be inconclusive was not anticipated going into the study. When checking safety of a composite FSP process, LTSA reports that the composite process deadlocks if some trace leads to either `Error` or `Stop`[4], and it returns a shortest such trace. However, when the composite FSP process is the parallel composition of an analysis

[4]primitive FSP processes which have no successors, but serve different purposes: `Stop` represents a deadlocked process, whereas `Error` serves as a trap state when checking properties.

model and a safety property (as is the case for the tables `LockProtocol`, `CorrectPairing` and `ExcessiveSignalling`), we can conclude that the analysis model does not satisfy the safety property only if the composite model can reach `Error`. Thus, in cases where LTSA returns a trace that leads to `Stop` rather than `Error`, we cannot conclude that the property does not hold. In fact, none of the candidate solutions used in this study violated the standard locking protocol; however, as shown in the table for `LockProtocol`, we could not verify this fact for 12 of the analysis models (the 12 that could deadlock). Note that this problem is an artifact of how LTSA checks safety properties, and is not intrinsic to model checking. This source of inconclusive results can be eliminated by adding an option designating a search for traces that lead to `Error` only.

Discussion

One question we strove to answer was whether the quality of candidate solutions can be judged using automated analysis of faithful models. In our study, we used these analyses to make a number of useful judgements. For instance, only participant No.3's submission can deadlock in a context in which a pair operation should never have to wait (client model `CRou`). Moreover, the submissions of the other participants pair the boys and girls correctly in this context. These observations provide an objective basis for ranking the quality of the submission of participant No.3 lower than that of the others, at least with regard to preventing deadlock, and possibly also with regard to correct pairing. Similarly, the submissions of participants No.3 and No.4 rank lower with regard to these same 2 properties than those submitted by No.1 and No.2 in the context where activations of different operations execute concurrently but activations of the same operation execute sequentially (client model `TPMe`). Regarding deadlock, the submission of participant No.1 ranks below that of participant No.2, as the former can deadlock in a context (represented by `CPai`) that may invoke pair operations concurrently. Regarding correct pairing, however, these latter two submission rank the same.

The results of checking conformance to the locking protocol provide no basis for ranking any candidate solution better than than any other. This is as it should be because every submission correctly conformed to this protocol. Regarding excessive signaling, the results indicate the submission of participant No.2 ranks higher than that of participant No.3, which in turn ranks higher than those of No.1 and No.4, and that these latter 2 submissions rank the same. We do not include the submission of participant No.5 in this ranking because of the inconclusive results. In contrast, because the submission of participant No.3 can exhibit excessive signaling in the context represented by `CRou`, it can also do so in the less restrictive contexts (`CPai` and `FCon`). Thus, we include this participant in the ranking, in spite of the dashed entries.

Another question we strove to answer was whether the generation of analysis models and properties is sufficiently systematic. We produced the client models and resource models manually, and then automatically assembled them into analysis models using the Noweb literate programming tool [23]. The resource models were systematically assembled from a library of reusable models of synchronization primitives and from simple models of program statements. For the candidate solutions produced by participants in our larger empirical study, this assembly process is certainly automatable. Additionally, given specifications of the numbers of threads and of the operations each thread performs, the client models could be automatically generated. To generate the analysis models for the pilot study, we manually created a Noweb document containing the resource models, the client models, and specifications for assembling them into analysis models. The assembly specifications were produced by instantiating a single template. Thus, we were able to systematically produce faithful FSP models and it would be feasible to automatically generate them for a larger follow-up study that uses the same problem.

That said, generating faithful FSP models of candidate solutions to general concurrent programming problems is not fully automatic. Substantial effort is required to create the problem-specific infrastructure needed to systematically generate the analysis models and

properties. Regarding the properties: No additional work is required to check for deadlock, but all the other checks require a property describing legal behaviors. The properties are reusable across all resource models created from the candidate solutions to the match-maker problem. This is to be expected, as a problem typically defines the properties that its solutions must satisfy.

We used FSP/LTSA for the pilot study because we felt FSP models could be both faithful to the code and suitably abstract. Faithfulness is important for ensuring the quality of the analysis model accurately reflects the quality of the code. Abstractness is necessary in order that exhaustive analysis is feasible. But abstractness can easily compromise faithfulness, so a balance is needed. Using tools, such as Java PathFinder [20] or Bandera [12], that extract models directly from source code, might save on effort needed to develop problem-specific infrastructure. We could not use these tools for our pilot study because our participants produced C++ code using synchronization primitives supplied by a library. In summary, we felt that FSP provided an appropriate balance.

For a larger study, the generation of the analysis models will have to be automated. Moreover, we will need to check properties of the models in batch mode and the results of the checks will need to be output in a processable form. Although we did not see how to run batch analyses or how to output traces or runs for subsequent processing with the version of LTSA available at [36], it should be possible to obtain a version of the tool with these capabilities.

5.3 Objective study: UML state machine modeling vs. UML sequence diagram modeling

Our previous studies showed that both UML state machine diagrams and UML sequence diagrams may help novices in the comprehension of concurrency and synchronization. We recognize that UML state machine diagrams and sequence diagrams are intended to be used in concert to depict the behavior of the modeling system. However, the question remains

whether the two diagrams differ significantly in their effectiveness in assisting such comprehension. To investigate this question and to tease apart the relative contribution of each type of diagram to the overall benefit derived from their use, we conducted a comparative, objective study of the benefits of each diagram when used by a group of student participants as aids to solve questions involving concurrency and synchronization. The effectiveness of each type of the diagram was measured by the corresponding group's performance in answering the questions.

Several challenges exist in conducting a fair comparison of UML state machine diagrams with UML sequence diagrams. UML state machine diagrams and UML sequence diagrams carry different information about the modeled system. A UML state machine diagram specifies the lifetime behavior of a single object. A set of UML state machine diagrams, each of which models a component object of the system, may cover the entire execution space of the modeled system. For example, to model all the behaviors of a readers-writer system, we need only three UML state machine diagrams that model, respectively, the shared monitor object *Database* (Figure 5.6 on page 72) and the two client objects *Reader* and *Writer* (Figure 5.11). In contrast, a single UML sequence diagram depicts only the thread interactions of one potential execution of the modeled system. A well-chosen collection of sequence diagrams can provide insight into key facets of the run-time behavior of the modeled system. However, due to the nondeterministic nature of thread scheduling, the number of potential executions of a concurrent program can often be too large to be reasonably depicted by a collection of UML sequence diagrams. Thus, the first challenge is how to choose a collection of UML sequence diagrams that covers all the interesting interactions of a modeled system. At the same time, we need to limit the number of UML sequence diagrams to a number that participants can handle within the time constraints of our study.

While UML state machine diagrams show a high-level overall view of the modeled system, UML sequence diagrams reveal the low-level implementation details of interactions between system components. When dealing with scenario-based questions involving the implementa-

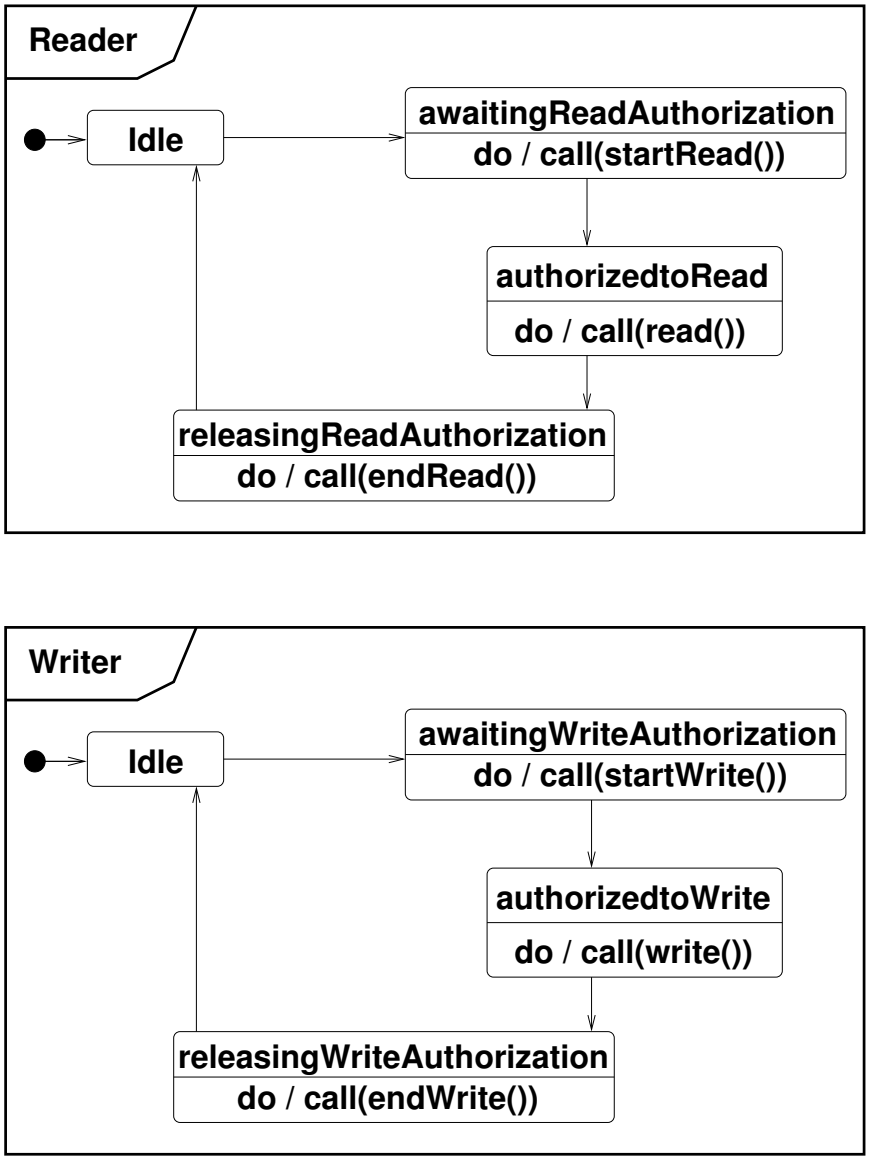


Figure 5.11: UML state machine models of *Reader* and *Writer*.

tion of synchronization logic, a user must make inferences based on the UML state machine diagrams. However, if the scenarios are directly depicted by the given UML sequence diagrams, a user can easily "read" the answers without making any inferences. Thus, a second challenge is how to present an appropriate collection of UML sequence diagrams so that they avoid directly depicting the question scenarios but still show the fundamental synchronization behaviors from which a user can infer the correct thread interactions of specific execution scenarios. Those two challenges reduce to a common problem: picking the "right" UML sequence diagrams. In the following section, we describe our approach to achieving this goal.

5.3.1 Using FSP models to generate candidate UML sequence diagrams

In this section, we briefly describe a technique of using formal models to generate a collection of UML sequence diagrams that cover all the key facets of the interaction space conveyed by state machines. Figure 5.12 presents a class diagram that describes this technique. The interaction space of a software system can be modeled in state machines, which may be defined as either FSP models or UML state machine diagrams. Further, an FSP model may be visualized by a flowchart, a graph of nodes connected by transitions. When each flowchart is a directed acyclic graph (DAG), we may find a set of paths that cover all the nodes in a flowchart. A path represents an execution of the system and can be depicted in a UML sequence diagram. Thus, we can generate a set of UML sequence diagrams that cover all the states defined in given FSP models. We consider this set of UML sequence diagrams to be "comparable" to the corresponding UML state machine diagrams that target the same state machines. We will illustrate this technique via the readers-writers example.

We recognize that without restricting the number of objects involved, the state space of a concurrent program can grow exponentially. Thus, we constrain our readers-writers model to allow only up to two *Reader* threads and two *Writer* threads competing for the access to the shared *Database*. We define four representative configurations that are capable of revealing

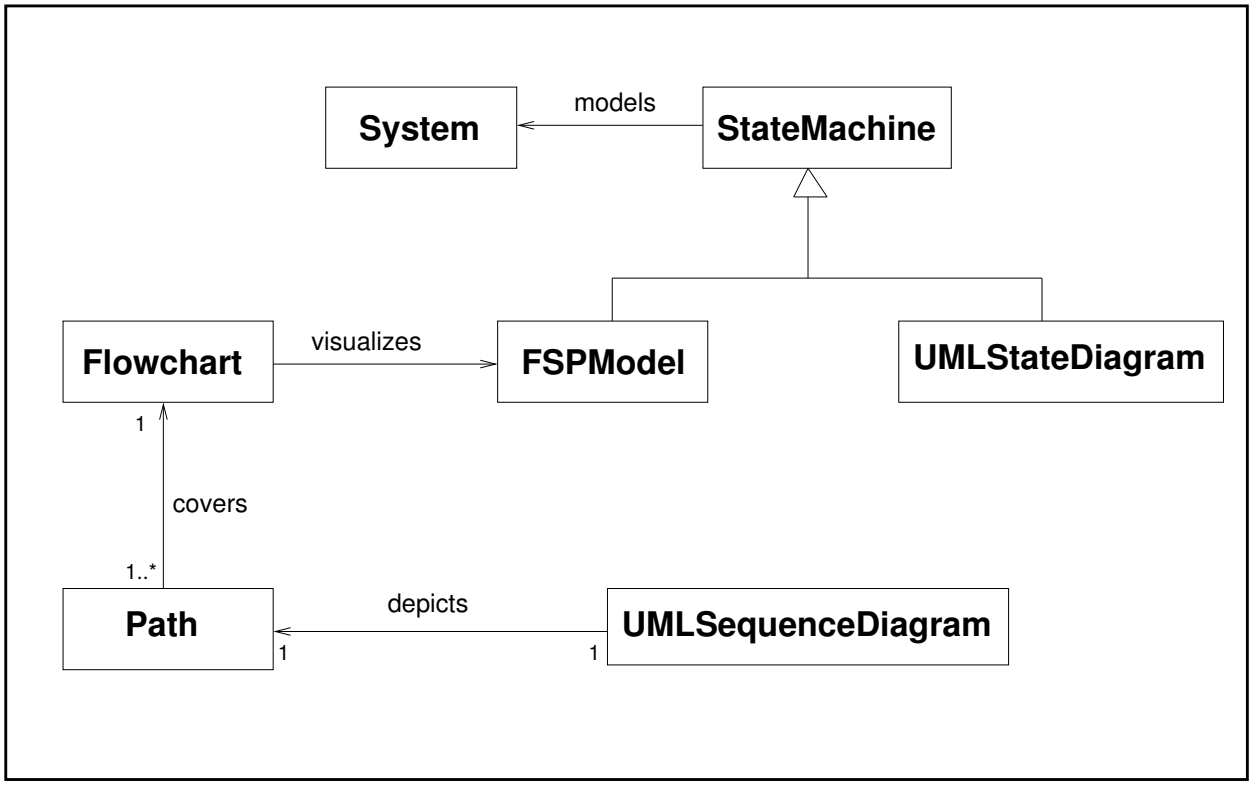


Figure 5.12: A technique for generating a collection of UML sequence diagrams that are comparable to a set of UML state machine diagrams, depicted in a class diagram

the essence of the synchronization behavior of the readers-writers problem. Table 5.5 summaries these four configurations.

Table 5.5: Four representative configurations

Name	Interpretation
TwoReaders	Interactions between two reader threads
TwoWriters	Interactions between two writer threads
ReaderAndWriter	Interactions between one reader thread and one writer thread
TwoReadersAndOneWriter	Interactions between two reader threads and one writer thread

Corresponding to each configuration, we created a composite FSP process. Figure 5.13 presents the FSP process `TWO_WRITERS` of the *TwoWriters* configuration. To limit state explosion, we adopt the following conventions in our FSP processes:

1. To keep each thread finite, an FSP process modeling a client thread allows only one visit to the shared *Database*. For instance, FSP process `WRITER_NR`, the sequential composition of the subprocess—`STARTWRITE`, `WRITE`, `STOPWRITE`, and `DONE`, models a writer thread that eventually terminates after a visit to *Database*. In practice, a client thread may visit the shared *Database* repeatedly.

2. We constrain the orderings of threads' method calls to elide "symmetric" scenarios. In Figure 5.13, the process `PREFER_W1_START` defines that the writer thread *w1* must call *startWrite* before the writer thread *w2*. This is symmetric to the scenario of *w2* calling *startWrite* first.

3. We treat the execution of a monitor method body as an atomic action. Thus, thread interleavings are prohibited inside of a monitor method body. Such optimization techniques can effectively reduce the number of interleavings and many unnecessary transitions and states in our FSP processes. For instance, in defining the process `STARTWRITE`, which models the monitor method *startWrite*, the action `incWrite` includes several operations—incrementation of the counter *numWriters*, release of the lock, and return

```
const NumReaders = 2
const NumWriters = 2
range ReaderRange = 0..NumReaders
range WriterRange = 0..NumWriters

DONE = ( done -> END ).
STARTWRITE = ( callStartWrite -> incWrite -> END ).
WRITE = ( write -> returnWrite -> END ).
STOPWRITE  = ( callStopWrite -> decWrite -> END ).
WRITER_NR = STARTWRITE; WRITE; STOPWRITE; DONE; END+{incRead, decRead}.

PREFER_W1_START = ( w1.callStartWrite -> W1_STARTED ),
W1_STARTED = ({w1.callStartWrite,w2.callStartWrite}->W1_STARTED | done->END ).

PREFER_W1_LOCK = ( w1.incWrite -> W1_LOCKED ),
W1_LOCKED = ( {w1.incWrite,w2.incWrite} -> W1_LOCKED | done -> END ).

PREFER_W1_WRITE = ( w1.write -> W1_WRITE ),
W1_WRITE = (w2.write->BOTH_WRITE | w1.{write, callStopWrite}->W1_WRITE ),
BOTH_WRITE = ( {w1,w2}.{write,callStopWrite} -> BOTH_WRITE | done -> END ).

PREFER_W1_STOP = ( w1.callStopWrite -> W1_STOPPING ),
W1_STOPPING = ({w1.callStopWrite, w2.callStopWrite}->W1_STOPPING | done->END).

RWDB = RW[0][0],
RW[nr:ReaderRange][nw:WriterRange] = ( when ((nr == 0) && (nw == 0))
                                        incWrite -> RW[nr][nw+1]
                    | when ((nw == 0) && (nr < NumReaders))
                          incRead -> RW[nr+1][nw]
                    | when (nr > 0)
                          decRead -> RW[nr-1][nw]
                    | when (nw > 0)
                          decWrite -> RW[nr][nw-1]
                    | done -> END ).

||TWO_WRITERS = ( w1:WRITER_NR/{done/w1.done} ||
                  w2:WRITER_NR/{done/w2.done} ||
                  {w1,w2}::RWDB/{done/w1.done,done/w2.done} ||
                  PREFER_W1_START ||
                  PREFER_W1_WRITE ||
                  PREFER_W1_STOP ||
                  PREFER_W1_LOCK )
                \{ w1.incRead, w2.incRead, w1.decRead, w2.decRead }.
```

Figure 5.13: FSP process of the *TwoWriters* configuration.

from the monitor method *startWrite*. Thus, our model allows no context switch to occur while a writer is executing `incWrite`. However, a potential context switch may occur prior to the execution of `incWrite` but after the execution of `callStateWrite`, which indicates the acquisition of the monitor mutex lock. In this case, the next running writer thread would block on the monitor mutex lock, as it is being held by another thread. Without this optimization, context switches may occur while a writer thread is executing the method body of *startWrite*. However, the next running writer thread would still block on the monitor mutex lock. That is, this optimization elides potential context switches that lead to the same thread synchronization behavior as depicted by the one context switch allowed in our model.

4. We elide the implementation details of synchronization primitives (*lock*, *wait*, etc.) to reduce the complexity of our FSP processes for analysis purposes. Still, the invocations of operations on the synchronization primitives are depicted in the UML sequence diagrams.

An FSP process can also be translated into a "flowchart" that visualizes its finite machine. In a flowchart, each node denotes a state of the modeled system and each transition is caused by an atomic action. As our four composite FSP processes are finite processes that include no loop, the corresponding flowcharts present as directed acyclic graphs. Figure 5.14 presents the flowchart of the FSP process `TWO_WRITERS`. Node 0 denotes the initial state of the system while node 13 denotes the ending state. Each path from node 0 to node 13 conveys a trace of actions, which represents a potential execution of the readers-writers system involving two writer threads, that trigger the state transitions along the path. For instance, Figure 5.15 presents the trace of actions conveyed in the path that travels though nodes 0–13 in Figure 5.14. Such an execution may also be visualized in a UML sequence diagram. Figure 5.16 presents a UML sequence diagram that describe the interactions between the two writer threads that conform to the trace depicted in Figure 5.15. Thus, if we find a collection of paths that covers all the nodes in a given flowchart, then we can create a set of UML

sequence diagrams, each depicting the execution represented by a path in that collection, that covers all the states of the corresponding FSP process.

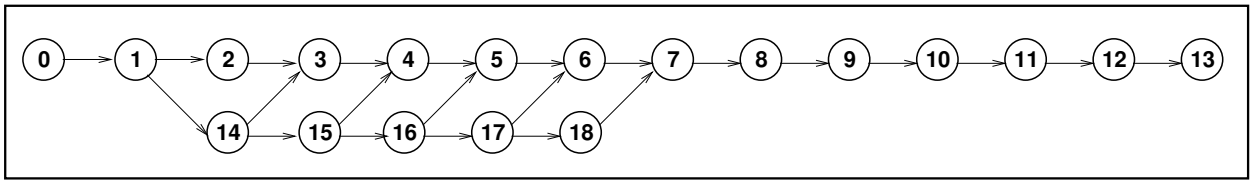


Figure 5.14: Flowchart representation of the state machine of `TWO_WRITERS`.

```
w1.callStartWrite
w2.callStartWrite
w1.incWrite
w1.write
w1.returnWrite
w1.callStopWrite
w1.decWrite
w2.incWrite
w2.write
w2.returnWrite
w2.callStopWrite
w2.decWrite
(done)
```

Figure 5.15: The trace of actions conveyed in the path that travels through nodes 0–13 in Figure 5.14.

By applying a Depth-First Search algorithm [9, sect. 22.3], all possible paths can be easily generated. Although the full collection containing all the paths is an obvious solution, it may contain many redundant paths. An ideal solution is a collection that contains the minimum number of paths that still cover all the nodes.

The minimum number of paths needed to cover all the nodes in a single-source/single-sink directed acyclic graph can be computed by the "minimum flow method" or the "maximum matching method" [41]. However, those two methods do not specify the exact paths included in the minimum path set.

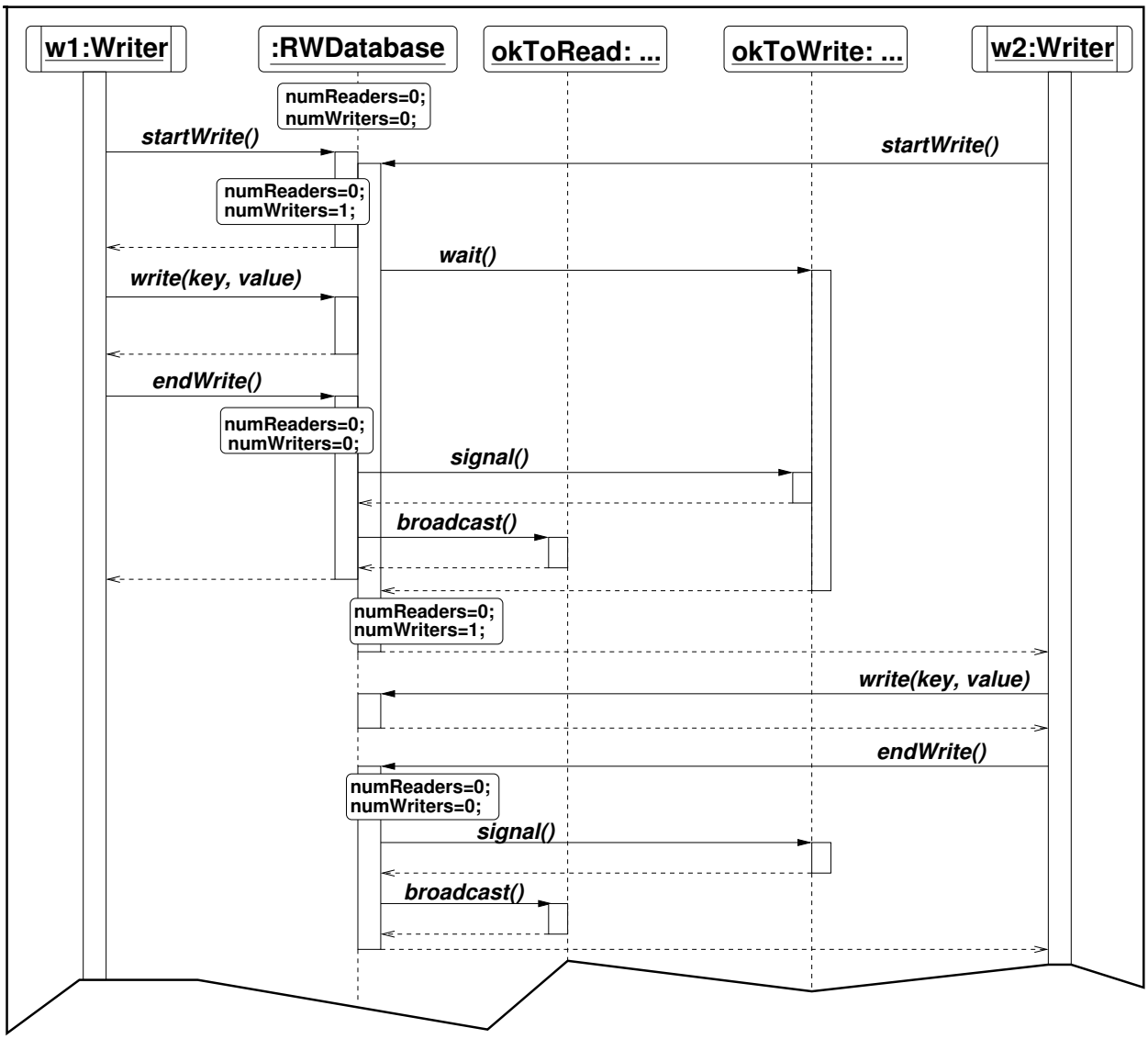


Figure 5.16: A UML sequence diagram for the *TwoWriters* configuration.

When we consider that each path is a *set* that contains the nodes on the path, then our question converts to determining the minimum number of sets so that every node is covered in at least one of the selected sets. This "set-covering" problem is one of the classical NP-complete problems confirmed by Richard M. Karp [25]. Such a problem can be solved by applying a greedy algorithm that is described in Figure 5.17.

```
Let V be the set of all nodes; S1, S2, ..., Sn are subsets of V.

A = {S1, S2, ..., Sn}
B = V
C = NULL
while B is not NULL {
   Select Si in A such that it has the largest number of nodes in B
   A = A - Si
   B = B - Si
   C = C union Si
}
return C
```

Figure 5.17: A greedy algorithm for the "set-covering" problem.

The flowchart in Figure 5.14 contains six distinct execution paths from the source node 0 to the sink node 13. Applying the greedy algorithm described in Figure 5.17, we can obtain the following two paths that together cover all the nodes:

- Path 1: 0→1→2→3→4→5→6→7→8→9→10→11→12→13
- Path 2: 0→1→14→15→16→17→18→7→8→9→10→11→12→13

Following this approach, we created nineteen UML sequence diagrams (see Appendix E.3) to cover all the states in the four FSP processes: five UML sequence diagrams for *TwoReaders*, two UML sequence diagrams for *TwoWriters*, four UML sequence diagrams for *ReaderAndWriter* and eight UML sequence diagrams for *TwoReadersAndOneWriter*. We claim that these nineteen UML sequence diagrams deliver equivalent information to that conveyed by the complete set of UML state machine diagrams that models the monitor-based solution of the readers-writers problem.

5.3.2 STUDY DESIGN

This study followed the same protocol—between subjects, pre-test/post-test—as used in our previous studies. Forty students from an undergraduate software design class at Michigan State University participated in the study in the spring of 2008. Students had studied the UML sequence diagram notation and the UML state machine diagram notation as part of the course material. Participants were considered novices to concurrency and synchronization. They had recently learned about concurrency and synchronization and were in the midst of implementing a small program that employed multiple threads using the ACE toolkit [49] and involved the implementation of a monitor construct and the use of condition synchronization. Their voluntary participation fulfilled a requirement for a homework grade, with students earning 0.625 points toward their final grade for each of the pre-test and the post-test. Students had the alternative option to complete a standard homework assignment. In addition, the participants also took a survey of user preferences after the post-test.

In the study, participants were divided into two equivalent groups based on their performance on the pre-test. Between the pre-test and the post-test, both groups attended a lecture during which the instructor reviewed of the use of both UML state machine diagrams and UML sequence diagrams as aids to the comprehension of concurrent programs. Then, in the post-test, one group, referred to as the *state* group, received the three UML state machine diagrams presented in Figure 5.6 and Figure 5.11 to assist their problem solving while the other group, referred to as the *sequence* group, received the nineteen UML sequence diagrams generated using the FSP models described in Section 5.3.1. A sample diagram from this set is seen in Figure 5.16. The effectiveness of the diagrams was measured by participants' performance on the post-test.

5.3.3 Session I

The first session of the study started with a lecture that briefly reviewed key concepts and terminologies of the UML modeling notations and concurrency involved in our study.

Examples of the use of UML lifeline states and concurrent composite states were introduced. The pre-test included twenty-one multiple-choice questions, seventeen of which appeared in the pre-test of the previous state modeling vs. non-modeling study. These seventeen questions tested participants' knowledge of concurrency and the UML state machine diagram notation. In addition, we added four questions about the UML sequence diagram notation. Please see Appendix E.1 for all the pre-test questions.

Based on participants' scores on the pre-test, we divided them into the state group and the sequence group with each group consisting of twenty participants, in an attempt to produce equivalent groups. The means and standard deviations of the two groups are presented in Table 5.6.

Table 5.6: State diagram vs. sequence diagram: pre-test results, normalized

	Mean	Standard deviation
The State Group	0.576	0.157
The Sequence Group	0.574	0.155

5.3.4 Session II

In the lecture component of the second session, the same instructor again reviewed the semantics of the UML state machine diagrams and how to use them to model monitor objects, using the bounded-buffer problem as an example (Figure 5.1). The instructor also emphasized the variability in the time at which a context switch may occur, which leads to the many different potential interactions between threads.

Following the lecture, participants took the post-test. The post-test consisted of twenty-six True/False questions, each of which required them to judge the feasibility of an event sequence involved in a given execution scenario. We deliberately designed those scenario-based questions so that none of the scenarios was directly depicted by any of the presented UML sequence diagrams. Thus, all of the True/False questions required inference for the

participants in both groups. Figure 5.18 presents an execution scenario involving two writer threads and the five True/False questions that appeared in the post-test. While the two presented UML sequence diagrams for the *TwoWriters* configuration depict that *w1* calls *startWrite* first, the problem scenario assumes that *w2* invokes *startWrite* first. Thus, participants in the sequence group cannot "read" answers directly from the two presented UML sequence diagrams.

```
Assume that only two threads exist in this system: w1 and w2. At the start
of the program writer thread w2 is running and has invoked the startWrite
method, followed by the write method. The write method call has returned.
A context switch occurs and the w1 thread begins to run.

Which of the following event sequences could happen next?
Circle YES if the event sequence is possible; otherwise, circle NO.

YES NO (a) w1 invokes startWrite and then blocks on the monitor lock.
YES NO (b) w1 invokes startWrite,followed by write,followed by endWrite.
YES NO (c) w1 invokes startWrite and then blocks because w2 has write
           authorization.
YES NO (d) w1 invokes startWrite and then blocks because w2 has write
           authorization. w2 calls endWrite, then startWrite, and write.
YES NO (e) w1 invokes startWrite and then blocks because w2 has write
           authorization. w2 calls endWrite. When the call to endWrite
           returns, w1 is in the ready state.
```

Figure 5.18: Sample questions appeared in the post-test.

In addition to the scenario-based questions, the post-test also included a coding question that required participants to fill in the method bodies of four monitor methods—*startRead*, *endRead*,*startWrite* and *endWrite*. The correct implementation of the four monitor methods requires participants to (1) place lock acquisition and release operations correctly in the code to ensure the monitor mutual exclusion; (2) use appropriate operations on the right condition variables under the correct guard conditions; and (3) update counting variables correctly. Appendix E.2 presents the post-test textual materials received by both groups.

Table 5.7: Post-test scores on the True/False questions, normalized

	Mean	Standard deviation
The State Group	0.689	0.136
The Sequence Group	0.658	0.211

5.3.5 Results and analysis

Table 5.7 summarizes the means and standard deviations of the two groups' scores on the twenty-six scenario-based True/False questions. No significant difference exists between the two groups based on the total scores of the twenty-six True/False questions. Although UML sequence diagrams, each of which describes an execution scenario, seem to be more informative than UML state machine diagrams when used as aids in solving scenario-based questions, the sequence group still slightly trailed the state group on the scores of the True/False questions. We suggest that this is due to the fact that none of the nineteen UML sequence diagrams received by the sequence group depicted a question scenario. The sequence group must infer the synchronization logic of the monitor-based implementation of the readers-writers problem from the nineteen UML sequence diagrams and then construct the behavior models of the question scenarios by themselves. Meanwhile, the state group also must infer the same synchronization logic from the three given UML state machine diagrams and construct the behavior models of the same question scenarios.

On a per-question basis, we found that the sequence group significantly outperformed the state group ($p < 0.05$) on a question that asked if, at the start of the program, a reader thread could invoke *read*, followed by *startRead*, followed by *endRead*. The correct answer is no, because a reader thread must invoke *startRead* to get the access authorization before it can invoke *read* to access the *Database*. This result conforms to the fact that UML sequence diagrams emphasize the time ordering of the operations while UML state diagrams do not.

In assessing the quality of the collected solutions to the coding question, we applied the formal modeling techniques discussed in Section 5.2.4. Each student solution was transcribed to a resource model. We then defined eight client models as summarized in Table 5.8. Each client model represents an FSP process modeling a distinct client configuration. These eight client models have a gradual increase in concurrency, ranging from a sequential client model (*OneReader* and *OneWriter*) to a representative concurrent client model (*TwoReadersAndTwoWriters*). The parallel composition of a resource model and a client model forms an analysis FSP model. Thus, each solution may generates eight analysis models.

We also defined four properties, summarized in Table 5.9, that are essential for the readers-writers problem. Each analysis model was checked against these four properties. The "lock-protocol" property checks if each access to shared objects is protected by correct lock acquisition and release operations. The "deadlock" property checks if a deadlock occurs. The "invariant" property checks if any of the three problem-specific invariant conditions are verified: the first invariant asserts that at any time the shared database can accommodate no thread, or only one writer thread, or one or multiple reader threads; the second invariant asserts that when a reader thread calls *read*, *numWriters* must be zero; the third invariant asserts that when a writer thread calls *write*, *numReaders* must be zero. The "excessive-signaling" property checks whether an FSP model based on a participant's solution generates more signals or broadcasts than the FSP model based on the correct solution.

Table 5.10 summarizes the means and standard deviations of the scores that the collected solutions obtained when they were verified against the four properties. No significant difference was found based on these scores. Further, we noticed that ten invalid solutions existed among the collected solutions. Four of the ten invalid solutions were collected from the state group and the other six were collected from the sequence group. For these invalid solutions, we found that several participants did not answer this question at all while the others only provided partial, severely defective answers that failed all the tests. We postulate that the occurrence of invalid solutions was largely due to a lack of motivation. In this

Table 5.8: Eight client models

Name	Interpretation
OneReader	one reader thread calls *startRead*, *read* and *endRead* in sequence
OneWriter	one writer thread calls *startWrite*, *write* and *endWrite* in sequence
TwoReaders	two reader threads, each calls its three methods in sequence
TwoWriters	two writer threads, each calls its three methods in sequence
ReaderAndWriter	one reader and one writer, each calls its three methods in sequence
ReaderAndTwoWriters	one reader and two writers, each calls its three methods in sequence
TwoReadersAndWriter	two readers and one writer, each calls its three methods in sequence
TwoReadersAndTwoWriters	two readers and two writers, each calls its three methods in sequence

Table 5.9: Four properties for code verification

Name	Interpretation
Lock-protocol	check if solutions conform to correct locking protocol
Deadlock	check if deadlock occurs
Invariants	check if invariant conditions are violated
Excessive-signaling	check if excessive signaling or broadcasting exists

study, participants could earn grade credits simply by attending the study. They understood that their performance on the tests would not affect their grades. Consequently, some of the participants did not answer the coding question, even though their good performance on the scenario-based True/False questions showed that they had the knowledge to handle questions related to concurrency and synchronization. The presence of solutions from participants who were not making an honest effort introduced noise into our analysis.

In order to clean the noise, we performed a peer-to-peer analysis in which we analyzed the scores of participants with top performance on the coding question in each group. Those participants were more likely to make an honest effort on the post-test. Table 5.11 and

Table 5.12 summarize the means and standard deviations of the scores of the participants whose total scores on the coding question fall in the top 25% and 50% of their groups, respectively. When considering the top 25% solutions, both a one-tailed, heteroscedastic t-test and a Wilcoxon rank-sum test show that the sequence group significantly outperformed the state group on the combination of the three total scores obtained for the coding question ($p < 0.02$). When considering the top 50% solutions, a one-tailed, heteroscedastic t-test shows the sequence group also significantly outperformed the state group on the combination of the three properties: deadlock, invariant, and excessive-signaling ($p < 0.04$). A Wilcoxon rank-sum test on those top 50% solutions also confirms this statistically significant result ($p < 0.03$).

Meanwhile, we notice that the state group actually scored slightly better than the sequence group on the lock-protocol property. This result is not surprising. While the UML sequence diagrams received by the sequence group did not show the lock-related operations, the UML state machine diagrams received by the state group conveyed the rendezvous semantics of mutual exclusion.

Table 5.10: Post-test mean/stdev for all solutions, normalized

	Lock protocol	Deadlock	Invariants	Excessive signaling
The State Group	0.288/0.431	0.494/0.432	0.481/0.339	0.722/0.381
The Sequence Group	0.250/0.444	0.569/0.469	0.483/0.402	0.616/0.435

Table 5.11: Post-test mean/stdev for top 25% solutions only, normalized

	Lock protocol	Deadlock	Invariants	Excessive signaling
The State Group	0.850/0.335	0.600/0.454	0.733/0.273	0.888/0.103
The Sequence Group	0.800/0.447	1.000/0.00	0.900/0.224	0.950/0.112

The survey of participants' preferences shows no strong correlation between participants' performance on the post-test and their preferences. Among the eighteen participants in

Table 5.12: Post-test mean/stdev for top 50% solutions only, normalized

	Lock protocol	Deadlock	Invariants	Excessive signaling
The State Group	0.525/0.506	0.750/0.373	0.629/0.281	0.888/0.097
The Sequence Group	0.500/0.527	0.838/0.354	0.800/0.258	0.938/0.102

the state group who took the survey, nine prefer the UML state machine diagram; eight prefer the UML sequence diagrams; one did not provide his preference. Among the seventeen participants in the sequence group who took the survey, nine prefer state machine diagrams; five prefer sequence diagrams; one does not like either; two like both equally.

To summarize, our study shows that UML state machine diagrams and UML sequence diagrams did not vary significantly in their effectiveness in assisting novices in solving scenario-based questions. However, UML sequence diagrams may benefit participants significantly over UML state machine diagrams when used as aids in coding concurrent programs. It is also possible that participants who have more experience with these types of diagrams may benefit more from UML state machine diagrams. In future studies, we will train participants so that they are more experienced with both diagrams prior to the studies.

We recognize several threats to validity in our study. A major threat is the lack of strong incentives for participants to complete the test. This threat introduced substantial noise into the analysis. In our future studies, we shall raise the incentives for participants to succeed. Another potential threat to validity concerns how well our questions differentiate participants' performance in the study. Both the state group and the sequence group scored around 67% on average on the True/False questions on the post-test. Considering that even a participant who randomly guessed the answers could get a score of 50%, our True/False questions may not be the ideal type of questions used in such an empirical study. Our future

studies shall choose multiple-choice questions over True/False questions on the tests. Other potential threats include the limited scale of questions and participants in our study.

Chapter 6

Conclusion and Future Work

6.1 Research contributions

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 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 conduct empirical studies to assess the effectiveness of various representations on human performance (and retention of knowledge) during comprehension, debugging and coding tasks involving concurrent programs.

To summarize, the major up-to-date contributions of our research include the following:

1. Investigated some of the major difficulties students encounter when learning how to use concurrency and synchronization primitives (Chapter 3).

2. Designed and evaluated the saUML sequence diagram notation to ease the identified difficulties. 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 (in programs that employ condition synchronization) counter variables. By making these

concepts explicit, saUML diagrams expose many of the otherwise invisible details at play during thread synchronization. Using empirical observation, we showed our saUML notation to be beneficial as compared to both a text-only presentation and the standard UML sequence diagram notation (Chapter 4).

3. Evaluated the usability of UML 2.0 state machine diagram notation when used as an aid in novices' comprehension of concurrency concepts. Our studies showed that the UML 2.0 state machine diagram notation aids in the comprehension of concurrency.

4. Applied a new method of using FSP models to compare UML state diagrams with UML sequence diagrams in a fair way (Chapter 5). An initial comparison between the efficacy of using UML state machine diagram notation to support the comprehension of currency and that of the UML sequence diagram notation showed UML sequence diagrams better support the comprehension of the functional logic of a multi-threaded program than UML state diagrams.

6.2 Future work

Several interesting research questions remain to be investigated. The most pressing of these concern whether the benefits of saUML are limited to programs with complex condition synchronization. In the first study that compared the saUML sequence diagram notation with the standard UML sequence diagram notation, participants using saUML outperformed those using standard UML, but the difference did not rise to the level of 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 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 will 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 saUML 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?

Moreover, we plan to formalize our saUML sequence diagram notation in the forms of UML 2.0 profiles [47]. Such profiles tailor our notation for modeling concurrent software and make it possible to introduce tool support for our notation.

We also recognize that several parallel state machine models at different levels of abstraction and traceability to code may exist for the same concurrent software system. A more abstract state machine model favors abstraction over traceability to code. It usually does not depict many of the operational details of thread synchronization using libraries such as ACE [49] or Pthreads [6]. For instance, in an abstract state machine model, a server object may enforce mutual exclusion by refusing to accept operation invocations when it is in a state that is part of a method body. Clients' requests are queued up at an entry until the server

object returns to a state that is not within the body of a method. In this case, the actual implementation of mutual exclusion using mutex locks is hidden in such an abstract state machine model. The state machine modeling *BoundedBuffer* in Figure 5.1(a) is an example of a more abstract state machine model. In contrast, a more implementation-traceable state machine model concedes abstraction to retain the ability to reveal the nuts and bolts of the implementation details of the code. We would expect to see the depiction of the operations on the monitor mutex and condition variables in such a model. However, a truly traceable state machine model is quite complex, especially when condition synchronization is involved and its learning curve is steep for novices. While a more abstract state machine model is needed to enable feasible exhaustive analysis, a more implementation-traceable (i.e. less abstract) state machine diagram may better reveal the dynamic mechanisms inherent within a concurrent software system. How to achieve balance between these competing concerns is also one of our research topics. As a first step, we conducted two studies to evaluate the usability of the more abstract UML state machine diagram notation (Chapter 5). In the future, we will evaluate the usability of the more implementation-traceable variants of UML state machine diagrams. A study comparing these two types of models may also be needed to reveal more insights that help to better balance the trade-offs between abstraction and traceability.

All of our prior studies were conducted to improve educational benefit. 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.

Bibliography

[1] "The vienna development method: The meta-language," D. Bjørner and C. B. Jones, Eds. London, UK: Springer-Verlag, 1978.

[2] J. Ammirati, M. Gerhardt, and D. Dye, "Using object-oriented thinking to teach ada," in *WADAS '90: Proceedings of the seventh Washington Ada symposium on Ada*. New York, NY, USA: ACM, 1990, pp. 277–300.

[3] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche, "The impact of UML documentation on software maintenance: An experimental evaluation," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 365–381, 2006.

[4] B. S. Bloom, *Taxonomy of Educational Objectives, Handbook I: Cognitive Domain*. New York: McKay, 1956.

[5] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A theory of communicating sequential processes," *J. ACM*, vol. 31, no. 3, pp. 560–599, 1984.

[6] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[7] S. Carr, J. Mayo, and C.-K. Shene, "ThreadMentor: a pedagogical tool for multithreaded programming," *J. Educ. Resour. Comput.*, vol. 3, no. 1, p. 1, 2003.

[8] S.-E. Choi and E. C. Lewis, "A study of common pitfalls in simple multi-threaded programs," in *Proc. 31st SIGCSE Tech. Symp. Comput. Sci. Educ. (SIGCSE 2000)*. New York, NY, USA: ACM, 2000, pp. 325–329.

[9] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[10] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"," *Commun. ACM*, vol. 14, no. 10, pp. 667–668, 1971.

[11] L. K. Dillon, R. E. K. Stirewalt, E. Kraemer, S. Xie, and S. D. Fleming, "Using formal models to objectively judge quality of multi-threaded programs in empirical studies," in *Workshop on Modeling in Software Engineering in ICSE' 08*, 2008.

[12] M. B. Dwyer *et al.*, "Tool-supported program abstraction for finite-state verification," in *Proc. of the International Conference on Software Engineering*, 2001, pp. 177–187.

[13] H. Erdogmus, "On the effectiveness of the test-first approach to programming," *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 226–237, 2005, member-Maurizio Morisio and Member-Marco Torchiano.

[14] C. Exton and M. Kölling, "Concurrency, objects and visualisation," in *ACSE '00: Proceedings of the Australasian conference on Computing education*. New York, NY, USA: ACM, 2000, pp. 109–115.

[15] H. Fecher, J. Schönborn, M. Kyas, and W. P. de Roever, "29 new unclarities in the semantics of UML 2.0 state machines." in *ICFEM*, ser. Lecture Notes in Computer Science, vol. 3785. Springer, 2005, pp. 52–65, copyright hold by Springer-Verlag. [Online]. Available: http://www.doc.ic.ac.uk/ hfecher/papers/Fecher05icfem.pdf

[16] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, L. K. Dillon, and S. Xie, "Refining existing theories of program comprehension during the maintenance for concurrent software," in *Proc. IEEE/ACM Int. Conf. on Program Comprehension. (ICPC 2008)*, 2008.

[17] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, S. Xie, and L. K. Dillon, "A study of student strategies for the corrective maintenance of concurrent software," in *Proc. IEEE/ACM Int. Conf. on Software Engineering (ICSE 2008)*, 2008.

[18] R. France, A. Evans, K. Lano, and B. Rumpe, "The uml as a formal modeling notation," *Comput. Stand. Interfaces*, vol. 19, no. 7, pp. 325–334, 1998.

[19] B. George and L. Williams, "An initial investigation of test driven development in industry," in *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2003, pp. 1135–1139.

[20] K. Havelund and T. Presburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, 1998.

[21] C. W. Higginbotham and R. Morelli, "A system for teaching concurrent programming," in *Proc. 22nd SIGCSE Tech. Symp. Comput. Sci. Educ. (SIGCSE 1991)*. New York, NY, USA: ACM, 1991, pp. 309–316.

[22] K. Jensen, *Coloured Petri nets: basic concepts, analysis methods and practical use, volume 3*. New York, NY, USA: Springer-Verlag New York, Inc., 1997.

[23] A. Johnson and B. Johnson, "Literate programming using noweb," *Linux J.*, p. 1, Oct. 1997.

[24] A. Karahasanović and R. C. Thomas, "Difficulties experienced by students in maintaining object-oriented systems: an empirical study," in *ACE '07: Proceedings of the ninth Australasian conference on Computing education*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2007, pp. 81–87.

[25] R. M. Karp, "Reducibility among combinatorial problems," pp. 85–103, 1972.

[26] Y. B.-D. Kolikant, "Learning concurrency: evolution of students' understanding of synchronization," *Int. J. Hum.-Comput. Stud.*, vol. 60, no. 2, pp. 243–268, 2004.

[27] J. Kramer, "Is abstraction the key to computing?" *Commun. ACM*, vol. 50, no. 4, pp. 36–42, 2007.

[28] S. Kuske, "A formal semantics of uml state machines based on structured graph transformation," in *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. London, UK: Springer-Verlag, 2001, pp. 241–256.

[29] M. Kutar, C. Britton, and T. Barker, "A comparison of empirical study and cognitive dimensions analysis in the evaluation of UML diagrams," in *Proc. 14th Psychology of Programming Interest Group*, 2002.

[30] L. Kuzniarz, M. Staron, and C. Wohlin, "An empirical study on using stereotype to improve understanding of UML models," in *Proc. 12th IEEE International Workshop on Program Comprehension*. Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 14 – 23.

[31] L. Lamport, "Concurrent reading and writing," *cacm*, vol. 20, no. 11, pp. 806–811, 1977.

[32] H. Leroux and C. Exton, "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)*, 2001.

[33] H. Leroux, A. Réquilé-Romanczuk, and C. Mingins, "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*. New York, NY, USA: Computer Science Press, Inc., 2003, pp. 201–206.

[34] J. Lilius and I. P. Paltor, "The semantics of uml state machines," Tech. Rep., 1999.

[35] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd ed. Wiley, 2006.

[36] J. Magee, J. Kramer, R. Chatley, and S. Uchitel, "Labelled transition system analyser." [Online]. Available: http://www.doc.ic.ac.uk/ltsa

[37] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 593–622, 1989.

[38] K. Mehner and A. Wagner, "Visualizing the synchronization of Java-threads with UML," in *Proc. 2000 IEEE Int. Symp. Visual Languages (VL 2000)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 199.

[39] M. Y. Ng and M. Butler, "Towards formalizing uml state diagrams in CSP," *sefm*, vol. 00, p. 138, 2003.

[40] O. Nierstrasz, "What is the 'object' in object-oriented programming?" in *Proceedings of the CERN School of Computing*, vol. CERN 87-04, Renesse, the Netherlands, 1986, pp. 43–53.

[41] S. C. Ntafos and S. L. Hakimi, "On path cover problems in digraphs and applications to program testing," *IEEE Trans. Softw. Eng.*, vol. 5, no. 5, pp. 520–529, 1979.

[42] I. Ober and I. Stan, "On the concurrent object model of UML," in *Proc. 5th Int. Euro-Par Conf. Parallel Process. (Euro-Par 1999)*. London, UK: Springer-Verlag, 1999, pp. 1377–1384.

[43] "EclipseUML," Omondo. [Online]. Available: http://www.omondo.com

[44] C. M. Pancake, "Visualization techniques for parallel debugging and performance tuning tools," Corvallis, OR, USA, Tech. Rep., 1994.

[45] C. A. Petri, "Kommunikation mit automaten." *New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377*, vol. 1, pp. 1–Suppl. 1, 1966, english translation.

[46] H. C. Purchase, M. McGill, L. Colpoys, and D. Carrington, "Graph drawing aesthetics and the comprehension of UML class diagrams: An empirical study," in *Proceedings, 2001 Asia-Pacific Symposium on Information Visualization*, vol. 9, 2001, pp. 129–137.

[47] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed. Addison–Wesley, 2004.

[48] M. Schader and A. Korthaus, "Modeling Java threads in UML," in *The Unified Modeling Language – Technical Aspects and Applications*, M. Schader and A. Korthaus, Eds. Physica-Verlag, Heidelberg, 1998, pp. 122–143. [Online]. Available: citeseer.ist.psu.edu/schader98modeling.html

[49] D. C. Schmidt and S. D. Huston, *C++ Network Programming: Systematic Reuse with ACE and Frameworks, Vol. 2.* Pearson Education, 2002, foreword By-Frank Buschmann.

[50] C. Shene and S. Carr, "The design of a multithreaded programming course and its accompanying software tools," 1998. [Online]. Available: citeseer.ist.psu.edu/shene98design.html

[51] J. M. Spivey, *The Z notation: a reference manual.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[52] P. Stevens, "UML and concurrency," in *Abstract State Machines*, 2003, pp. 151–165.

[53] R. E. K. Stirewalt, "On the usability of UML 2.0 state diagrams for modeling and reasoning about the behavior of multi-threaded programs," Personal communication, 2007.

[54] J. Swan, T. Barker, C. Britton, and M. Kutar, "An empirical study of factors that affect user performance when using uml interaction diagrams," in *Proceedings, 2005 International Symposium on Empirical Software Engineering*, 2005, p. 10.

[55] A. J. Symonds, "Creating a software engineering knowledge base," in *An international workshop on Advanced programming environments.* London, UK: Springer-Verlag, 1986, pp. 494–506.

[56] S. Tilley and S. Huang, "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*, 2003, pp. 184 – 191.

[57] M. Torchiano, "Empirical assessment of UML static object diagrams," in *Proc. 12th IEEE International Workshop on Program Comprehension*. Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 226 – 230.

[58] Y. Wang, J.-P. Talpin, A. Benveniste, P. L. Guernic, and I. Irisa, "A semantics of uml state-machines using synchronous pre-order transition systems," in *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2000, p. 96.

[59] S. Xie, E. Kraemer, and R. E. K. Stirewalt, "Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts," in *Proc. 29th Int. Conf. Software Eng. (ICSE 2007)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 727–731.

[60] S. Xie, E. Kraemer, R. E. K. Stirewalt, L. K. Dillon, and S. D. Fleming, "Assessing the benefits of synchronization-adorned sequence diagrams: two controlled experiments," in *SoftVis '08: Proceedings of the 2008 ACM symposium on Software visualization*. New York, NY, USA: ACM, 2008.

[61] S. Xie, E. T. Kraemer, and R. E. K. Stirewalt, "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)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 123–134.

[62] W. L. Yeung, K. R. P. H. Leung, J. Wang, and W. Dong, "Improvements towards formalizing uml state diagrams in csp," in *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer

Society, 2005, pp. 176–184.

[63] J. Zhang and D. Norman, "Representations in distributed cognitive tasks," *Cognitive Science*, pp. 87–122, 1994.

Appendix A

Subjective survey questions

Question 1: Have you ever studied the UML sequence-diagram notation before?

Please respond to questions 2-5 by using the following rating scale: 1 strongly disagree; 2 moderately disagree; 3 undecided; 4 moderately agree; 5 strongly agree.

Question 2: this kind of sequence diagram is helpful in clarifying threads entering and exiting a monitor routine.

Question 3: this kind of sequence diagram is helpful in clarifying when and which threads are actively running on the processor at any given time (we assume multiple threads share a single processor).

Question 4: this kind of sequence diagram is helpful in illustrating the interactions between threads in a single program trace.

Question 5: this kind of sequence diagram is helpful in facilitating my understanding of the inherent mechanisms of monitors.

Please provide your comments in questions 6-7:

Question 6: Can you think of any other aspects of concurrency and/or synchronization behavior that this variant of the sequence-diagram notation might clarify or any ways in which it might aid in design, understanding, or verification activities? Please list any such aspects and briefly explain how/why diagrams in this notation might prove useful.

Question 7: Can you think of any aspects of concurrency and/or synchronization behavior that this variant of the sequence-diagram notation might obfuscate or that might complicate design, understanding, or verification activities? Please list any such aspects and briefly explain what it is about this notation that might lead to these problems.

Appendix B

Materials used in the saUML vs. text-only study

B.1 saUML vs. text-only: Pre-test questions

Name:_____

Q1. Link the correct descriptions to the terms:

Thread	The current thread changes from running to ready while one thread from the ready queue changes from ready to running.
Context switch	Sequential stream of execution.
Race condition	Section of code that must be executed atomically, i.e. by one thread at a time.
Atomic operation	A class-like programming language construct within which only one thread may execute concurrently.
Critical section	Computation depends on ordering of thread execution.
Deadlock	A situation where only one thread may access a resource at a time.
Mutual exclusion	Operation that must be performed entirely or not performed at all.
Monitor	A situation in which no progress can be made

Q2. You implement monitors. Your friend writes a concurrent program that shows that while Thread *t1* is in this monitor, it is context switched out and Thread *t2* runs. Your friend says that your monitor implementation is wrong. Is your friend correct?

1. Your friend is right.
2. Your friend had a wrong observation. *t1* can't possibly context switch out while holding a monitor.
3. Your friend is wrong. He must have designed a defective concurrent program.
4. Your friend is wrong. Although *t1* is switched out and *t2* runs, *t2* still can't get into the monitor.

Bank Account Problem:

```
class BankAccount extends Object {

   private double balance = 0;
   private ConditionVariable OKtoWithdraw = new
ConditionVariable();

   public synchronized void deposit(double amount) {
      balance = balance+amount;
      notifyAll(OKtoWithdraw);
   }

   public synchronized void withdraw(double amount) {
      while (amount > balance)
         wait(OKtoWithdraw);
      balance = balance-amount;

   }

}
```

For the following scenarios, we assume that only two **customer** threads (c1 and c2) are running on the processor.

Scenario 1:
Assume the current balance is 0. c1 is running within the invocation of *deposit(100)* and c2 is in the *ready* state. A context switch occurs. c2 changes to running and c1 changes to ready. c2 invokes *withdraw(150)*, but suspends afterwards.

```
public synchronized void deposit(double amount) {
      //context switch occurs here.
      balance = balance+amount;
      notifyAll(OKtoWithdraw);
}
```

Q3: Why does c2 suspend?

1. c2 suspends on the conditional variable OKtoWithdraw, since it wants to withdraw more money (150) than the current balance (0);
2. Deadlock occurs. Since c1 is in the monitor, c2 can't possibly enter the monitor;
3. c2 suspends on the monitor lock, since c1 is still holding the lock;

Q4: As a result:

1. c1 remains *ready* and c2 remains *running* and eventually finishes its withdraw transaction;
2. c1 changes to *suspended* and c2 remains *running* and eventually finishes its withdraw transaction;

3. c1 changes to *running* and c2 changes to *ready*;
4. c1 changes to *running* and c2 changes to *suspended*;
5. Both c1 and c2 change to *suspended*, since deadlock occurs.

Scenario 2:
Assume c1 is running within the invocation of *deposit(100)* and c2 is in the *suspended* state (we assume c2 was previously suspended on the monitor lock). c1 issues a *notifyAll(OKtoWithdraw)*.

```
public synchronized void deposit(double amount) {
      balance = balance+amount;
      notifyAll(OKtoWithdraw); // c1 implements this
}
```

Q5: As a result:
1. c1 changes to *ready* and c2 changes to *running*;
2. c1 changes to *suspended* and c2 changes to *running*;
3. c1 remains *running* and c2 remains *suspended*;
4. c1 remains *running* and c2 changes to *ready*;
5. Deadlock occurs.

Scenario 3:
Assume c1 is running within the invocation of *deposit(100)* and c2 is in the *suspended* state (it was suspended on the monitor lock).

Q6: If c1 releases the monitor lock and leaves the monitor, then:
1. c1 changes to *ready* and c2 changes to *running*;
2. c1 changes to *suspended* and c2 changes to *running*;
3. c1 remains *running* and c2 remains *suspended*;
4. c1 remains *running* and c2 changes to *ready*;
5. Deadlock occurs.

Scenario 4:
Assume the current balance is 100. c1 is in the *ready* state and c2 is running within the invocation of *withdraw(150)*. Since c2 wants to withdraw 150 and the current balance is only 100, c2 issues a *wait(OKtoWithdraw)*.

```
public synchronized void withdraw(double amount) {
      while (amount > balance)
         wait(OKtoWithdraw); // c2 implements this
      balance = balance-amount;
}
```

Q7. As a result:
1. c1 remains *ready* and c2 remains *running*;
2. c1 changes to *suspended* and c2 remains *running*;

3. c1 changes to *running* and c2 changes to *suspended*;
4. c1 changes to *running* and c2 changes to *ready*;
5. Deadlock occurs, since c2 suspends inside of the monitor. c1 will never be able to enter the monitor.

Scenario 5:
Assume the initial balance is 100. c1 is in the *running* state and c2 is in the *suspended* state (we assume c2 was previously suspended on *OKtoWithdraw*). c1 invokes deposit(100) and enters the monitor. It updates the balance to 200 and issues a *notifyAll(OKtoWithdraw).*

Q8: As a result:
1. c1 changes to *ready* and c2 changes to *running*;
2. c1 changes to *suspended* and c2 changes to *running*;
3. c1 remains *running* and c2 remains *suspended*;
4. c1 remains *running* and c2 changes to *ready*;
5. Deadlock occurs.

B.2 saUML vs. text-only: Post-test questions

NAME:_____

Readers-Writers Problem:

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

```
class Database extends Object {
   private int numReaders = 0;
   private int numWriters = 0;
   private ConditionVariable OKtoRead = new ConditionVariable();
   private ConditionVariable OKtoWrite = new ConditionVariable();

   public synchronized void startRead() {
      while (numWriters > 0)
           wait(OKtoRead);
      numReaders++;
   }
   public synchronized void endRead() {
      numReaders--;
      notify(OKtoWrite);
   }
   public synchronized void startWrite() {
      while (numReaders > 0 || numWriters > 0)
           wait(OKtoWrite);
      numWriters++;
   }
   public synchronized void endWrite() {
      numWriters--;
      notify(OKtoWrite);
      notifyAll(OKtoRead);
   }
}
class Reader extends Object implements Runnable {
   private Monitor m = null;
   public Reader(Monitor m) {
      this.m = m;
      new Thread(this).start();
   }
   public void run() {
      //do something;
      m.startRead();
      //do some reading…
      m.endRead();
      // do something else for a long time;
   }
}
class Writer extends Object implements Runnable {
   private Monitor m = null;
   public Writer(Monitor m) {
      this.m = m;
      new Thread(this).start();
   }
   public void run() {
      //do something;
      m.startWrite();
      //do some writing…
      m.endWrite();
      // do something else for a long time;

   }
}
```

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

For the following scenarios, we assume that only one **reader** thread and one **writer** thread are running on the processor.

Scenario 1:
Assume the **reader** thread is running within the invocation of *startRead()* and the **writer** thread is in the *ready* state. A context switch occurs just after the **reader** thread increments *numReaders* by one. The **reader** thread changes to *ready* and the **writer** thread changes to *running*. The **writer** thread invokes *startWrite()*.

```
public synchronized void startRead() {
   while (numWriters > 0)
       wait(OKtoRead);
   numReaders++;
  //context switch occurs here.
}
```

Q1. What will happen next:

A. The **reader** thread remains *ready*; the **writer** thread remains *running* and enters the monitor.
B. The **reader** thread changes to *suspended*; the **writer** thread remains *running*.
C. The **reader** thread changes to *running*; the **writer** thread changes to *ready*.
D. The **reader** thread changes to *running*; the **writer** thread changes to *suspended*.
E. Both the **reader** thread and the **writer** thread change to *suspended* and deadlock occurs.

Scenario 2:
Assume the **reader** thread is in the *running* state and the **writer** thread is in the *suspended* state (we assume the **writer** thread was previously suspended on *wait(OKtoWrite*). The reader thread invokes *endRead()* and enters the monitor. It sets the *numReaders* to 0 and issues a *notify(OKtoWrite)*.

Q2. As a result:

A. The **reader** thread changes to *ready*; the **writer** thread changes to *running*.
B. The **reader** thread changes to *suspended*; the **writer** thread changes to *running*.
C. The **reader** thread remains *running*; the **writer** thread remains *suspended*.
D. The **reader** thread remains *running*; the **writer** thread changes to *ready*.
E. Deadlock occurs.

Scenario 3:

Assume the **reader** thread is reading the *Database* and the **writer** thread is in the *ready* state. A context switch occurs. The **reader** thread changes to *ready* and the **writer** thread changes to *running*. The **writer** thread issues a *startWrite()*.

Q3. What will happen as a result of this invocation of *startWrite()*?

A. The **reader** thread remains *ready*; the **writer** thread remains *running* until the invocation completes.
B. The **writer** thread remains *running*; the **reader** thread changes to *suspended*.
C. The **writer** thread changes to *ready*; the **reader** thread changes to *running*.
D. The **writer** thread changes to *suspended*; the **reader** thread changes to *running*.
E. Deadlock occurs, since the **reader** thread is in the *Database* and the **writer** thread suspends inside of the monitor.

Scenario 4:
Assume the **writer** thread is running within the invocation of *startWrite()* and the **reader** thread is in the *ready* state. A context switch occurs just after the **writer** thread increments *numWriters* by one. The **Writer** thread changes to *ready* and the **reader** thread changes to *running*. The **reader** thread invokes *startRead()*, but suspends afterwards.

```
public synchronized void startWrite() {
   while (numReaders > 0 || numWriters > 0)
      wait(OKtoWrite);
    numWriters++;
   //context switch occurs here.
}
```

Q4. Why does the **reader** thread suspend?

A. The **reader** thread suspends on *wait(OKtoWrite*), since *numWriters* is non-zero at the time;
B. Deadlock occurs. Since the **writer** thread is in the monitor, the **reader** thread can't possibly enter the monitor;
C. The **reader** thread suspends on the monitor lock;
D. The **reader** thread suspends on *wait(OKtoRead*), since *numWriters* is non-zero at the time.

Scenario 5:
Assume the **reader** thread is running within the invocation of *startRead()* and the **writer** thread is in the *suspended* state (it was suspended on the monitor lock).

Q5. If the **reader** thread releases the monitor lock and leaves the monitor, then:

A. The **reader** thread changes to *ready*; the **writer** thread changes to *running*.
B. The **reader** thread changes to *suspended*; the **writer** thread changes to *running*.
C. The **reader** thread remains *running*; the **writer** thread remains *suspended*.
D. The **reader** thread remains *running*; the **writer** thread changes to *ready*.
E. Deadlock occurs.

Scenario 6:

Assume the **writer** thread is writing the *Database* and the **reader** thread is in the *ready* state. A context switch occurs. The **writer** thread changes to *ready* and the **reader** thread changes to *running*. The **reader** thread issues a *startRead()*, but suspends afterwards.

Q6. Why does the **reader** thread suspend?

A. The **reader** thread suspends on *wait(OKtoWrite*).
B. Deadlock occurs.
C. The **reader** thread suspends on the monitor lock.
D. The **reader** thread suspends on *wait(OKtoRead*).

Scenario 7:
Assume the **reader** thread is running within the invocation of *endRead()* and the **writer** thread is in the *suspended* state (the **writer** thread was previously suspended on the monitor lock). The **reader** thread invokes *notify(OKtoWrite).*

Q7. As a result:

A. The **reader** thread changes to *ready*; the **writer** thread changes to *running*.
B. The **reader** thread changes to *suspended*; the **writer** thread changes to *running*.
C. The **reader** thread remains *running*; the **writer** thread remains *suspended*.
D. The **reader** thread remains *running*; the **writer** thread changes to *ready*.
E. Deadlock occurs.

Q8. What feature(s) of the monitor implementation prevent race conditions in updating counting variables (*numReaders* and *numWriters*) ?

A. *wait* on condition variables.
B. *notify* on condition variables.
C. Lock is released before a thread suspends in wait.
D. The implicit lock on the monitor.

Q9. What feature(s) of the monitor implementation prevent a **writer** from entering the *Database* while **reader(s)** are present?

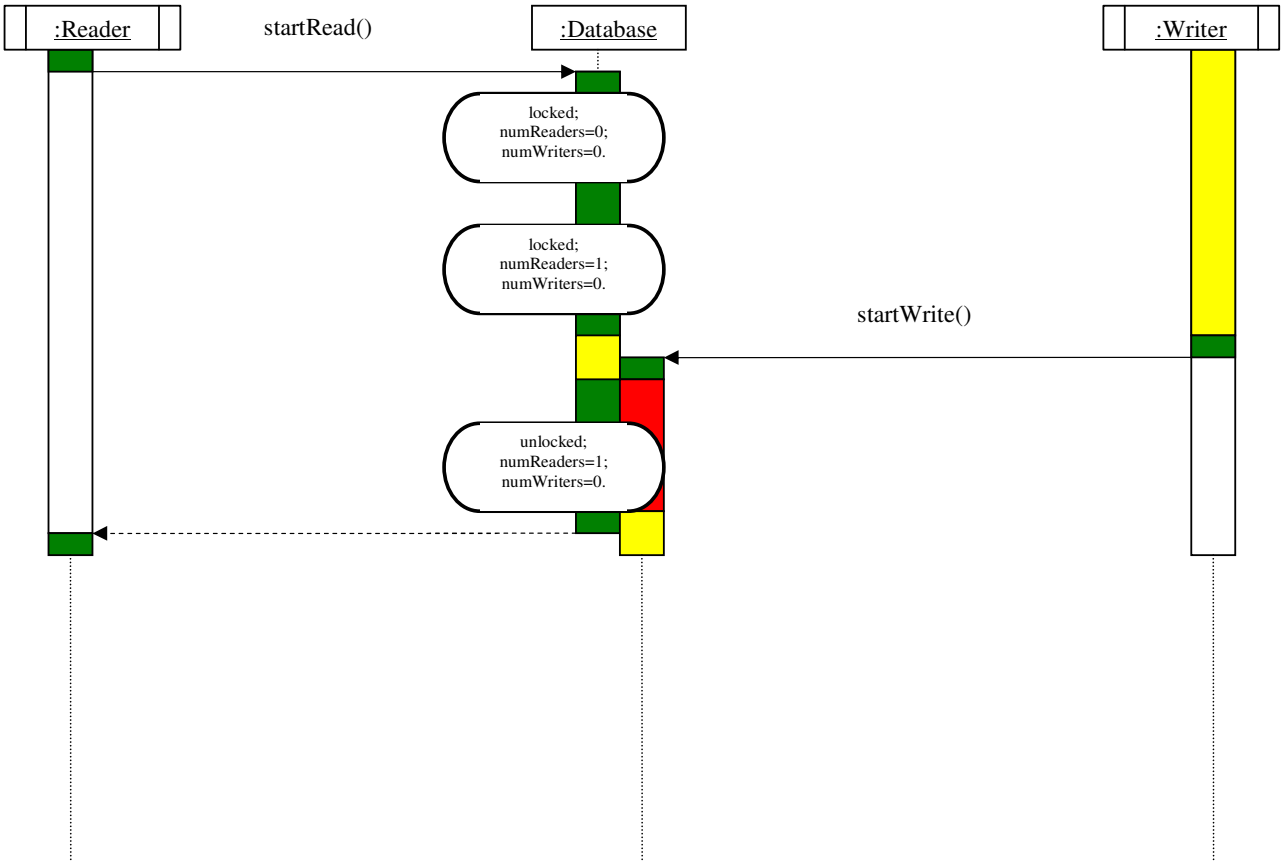
A. *wait* on condition variables.
B. *notify* on condition variables.
C. Lock is released before a thread suspends in wait.
D. The implicit lock on the monitor.

Q10. Why does the *wait* method release the lock and then acquire it again?

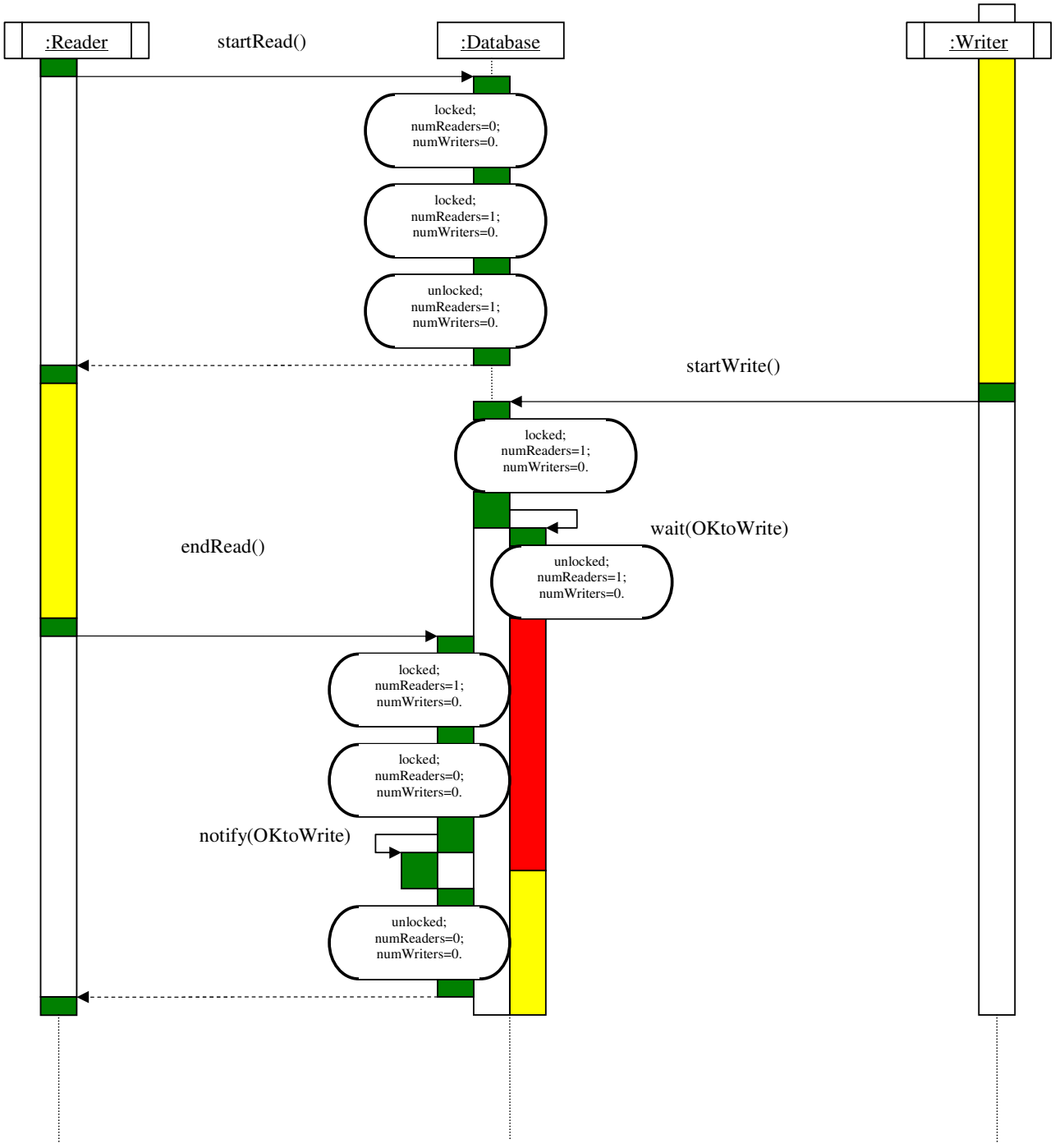
A. To “wake-up” a **reader** thread that was previously blocked on the *wait*.
B. To “wake-up” a **writer** thread that was previously blocked on the *wait*.
C. To promote efficiency.
D. To prevent deadlock.

B.3 Post-test saUML diagrams provided to the treatment group

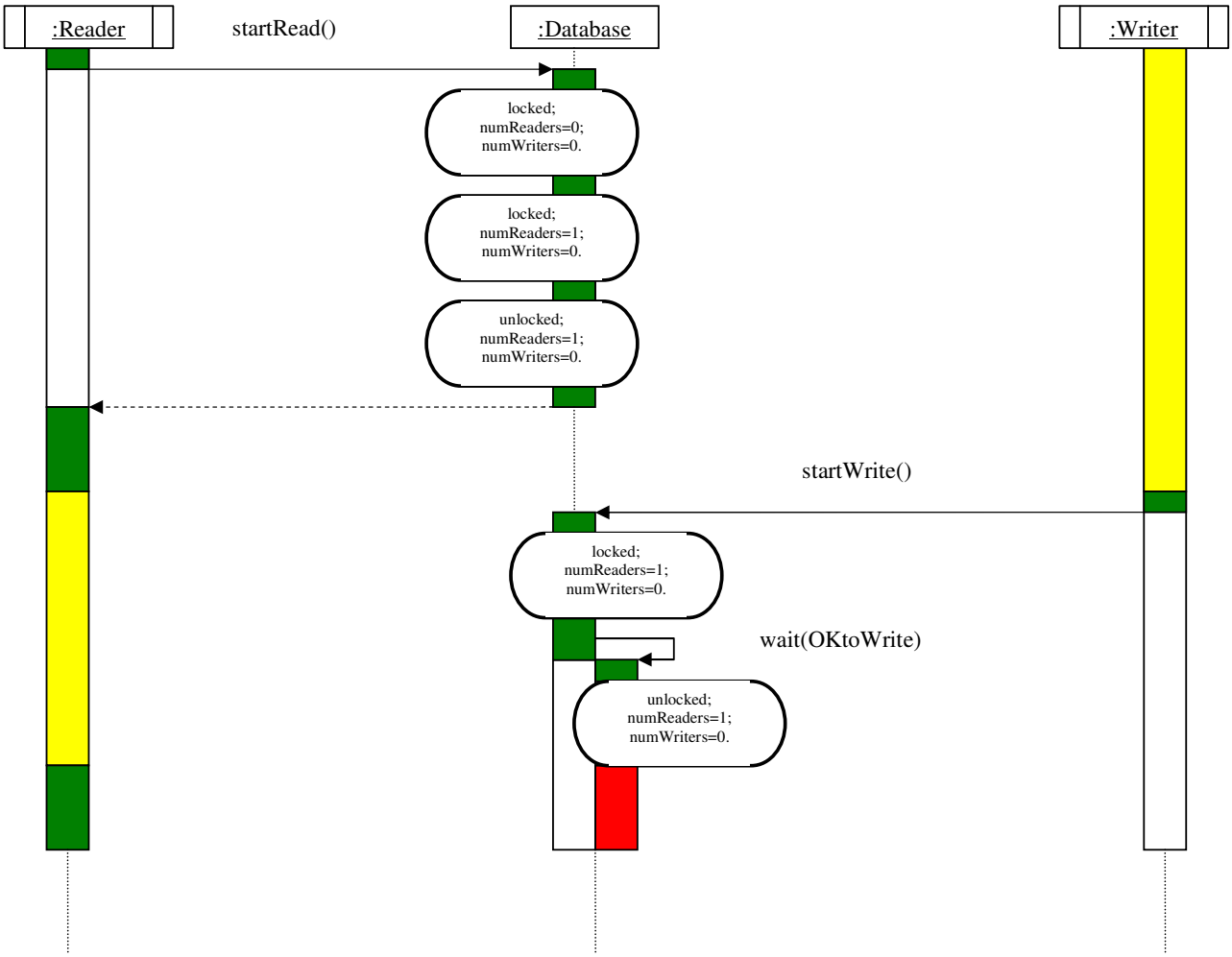
Scenario 1:


:Reader
startRead()
:Database
:Writer
locked;
numReaders=0;
numWriters=0.
locked;
numReaders=1;
numWriters=0.
startWrite()
unlocked;
numReaders=1;
numWriters=0.

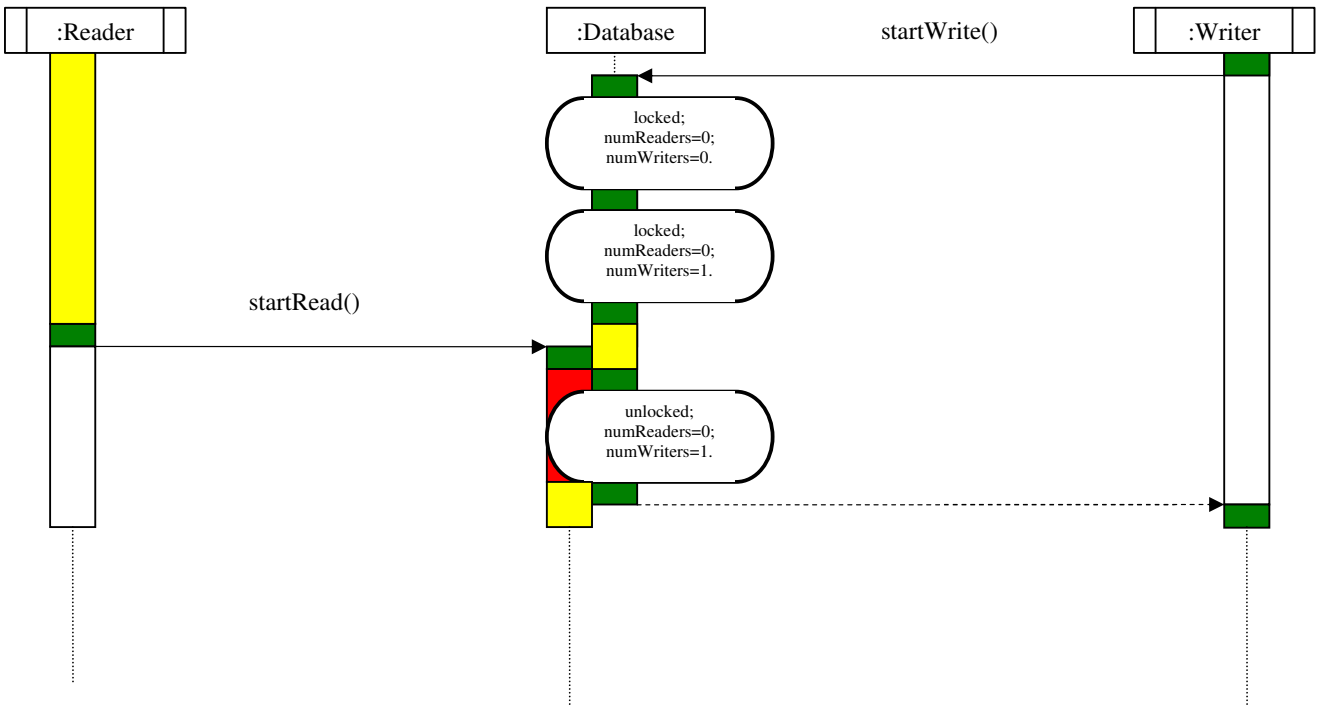
Scenario 2


:Reader
startRead()
:Database
:Writer
locked;
numReaders=0;
numWriters=0.
locked;
numReaders=1;
numWriters=0.
unlocked;
numReaders=1;
numWriters=0.
startWrite()
locked;
numReaders=1;
numWriters=0.
wait(OKtoWrite)
endRead()
unlocked;
numReaders=1;
numWriters=0.
locked;
numReaders=1;
numWriters=0.
locked;
numReaders=0;
numWriters=0.
notify(OKtoWrite)
unlocked;
numReaders=0;
numWriters=0.

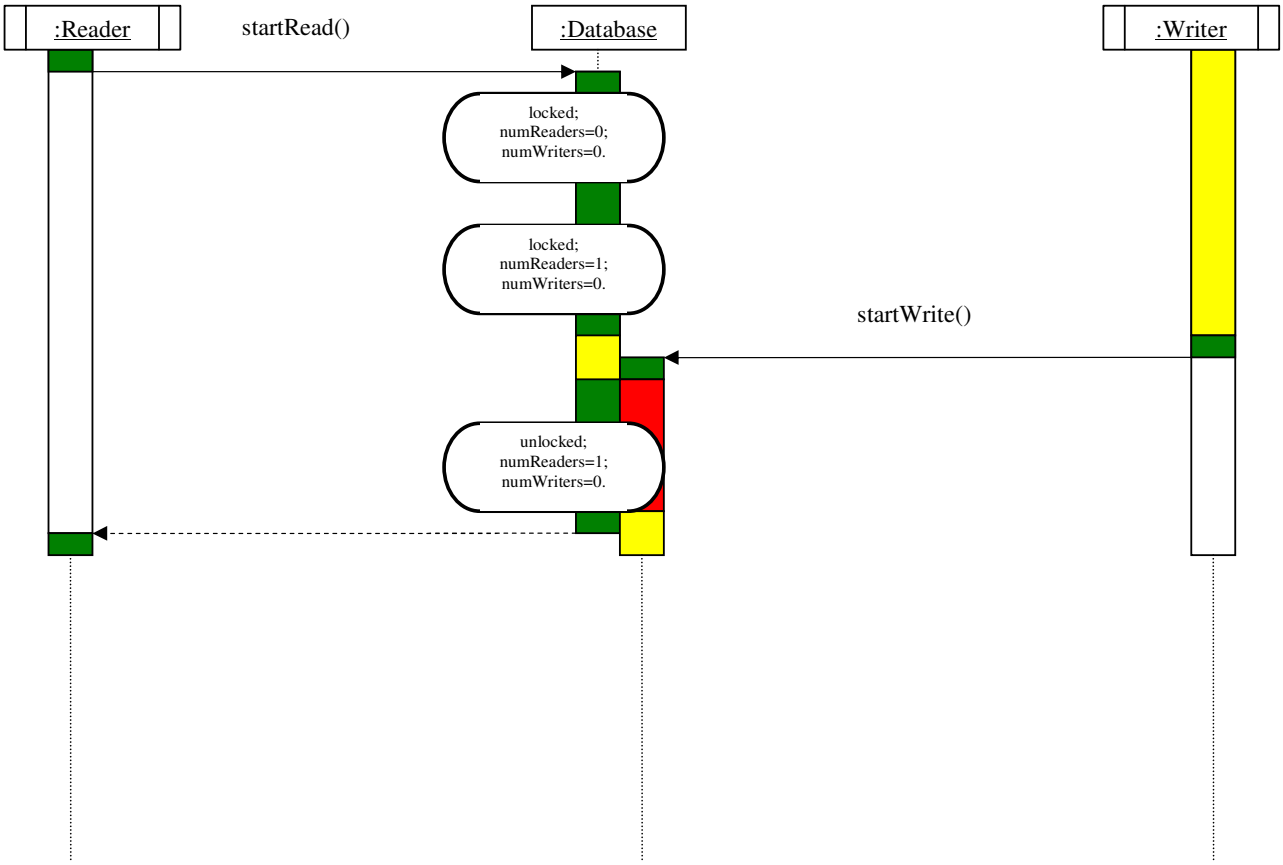
Scenario 3


:Reader
startRead()
:Database
:Writer
locked;
numReaders=0;
numWriters=0.
locked;
numReaders=1;
numWriters=0.
unlocked;
numReaders=1;
numWriters=0.
startWrite()
locked;
numReaders=1;
numWriters=0.
wait(OKtoWrite)
unlocked;
numReaders=1;
numWriters=0.

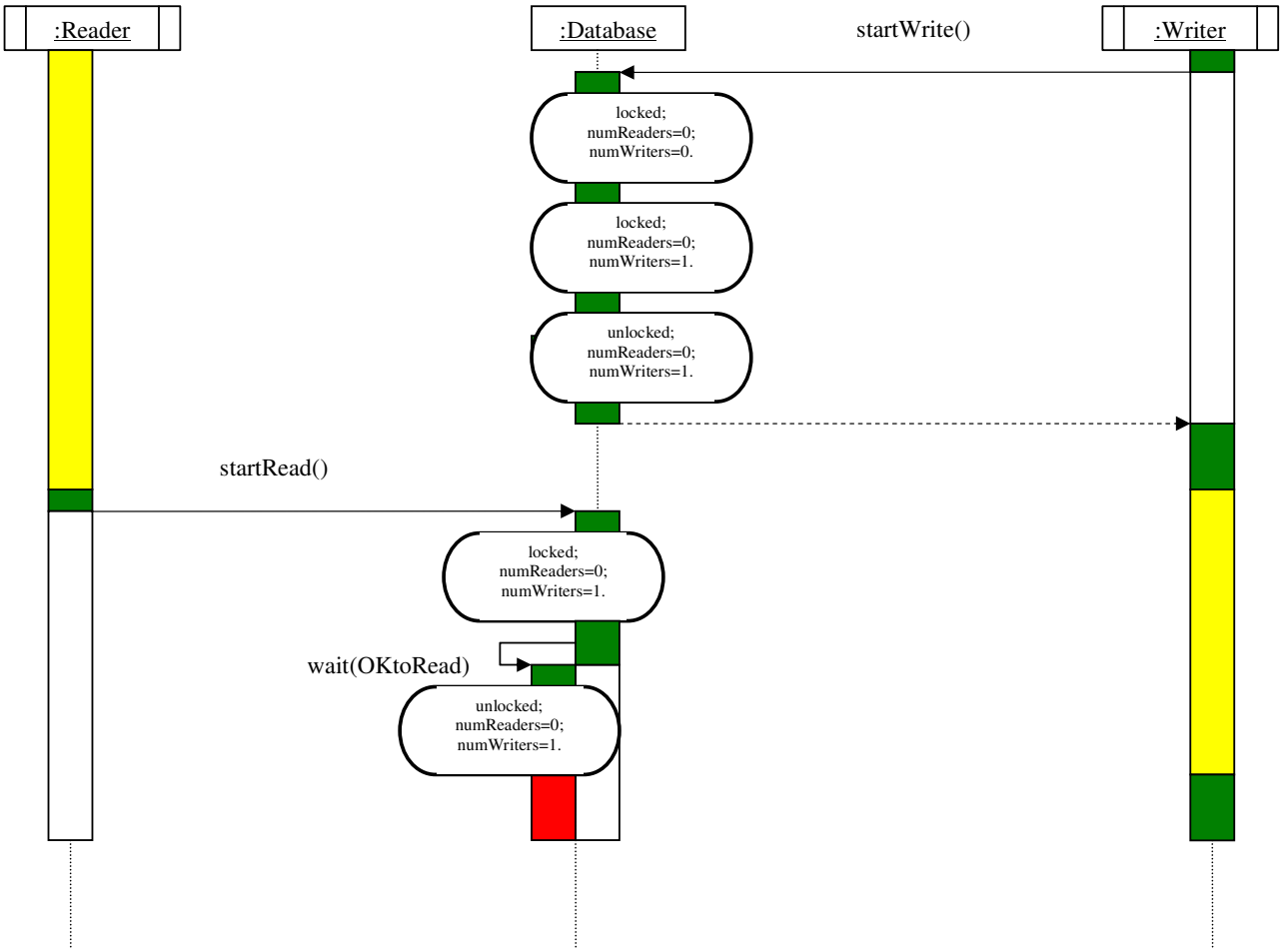
Scenario 4


:Reader
:Database
startWrite()
:Writer
locked;
numReaders=0;
numWriters=0.
locked;
numReaders=0;
numWriters=1.
startRead()
unlocked;
numReaders=0;
numWriters=1.

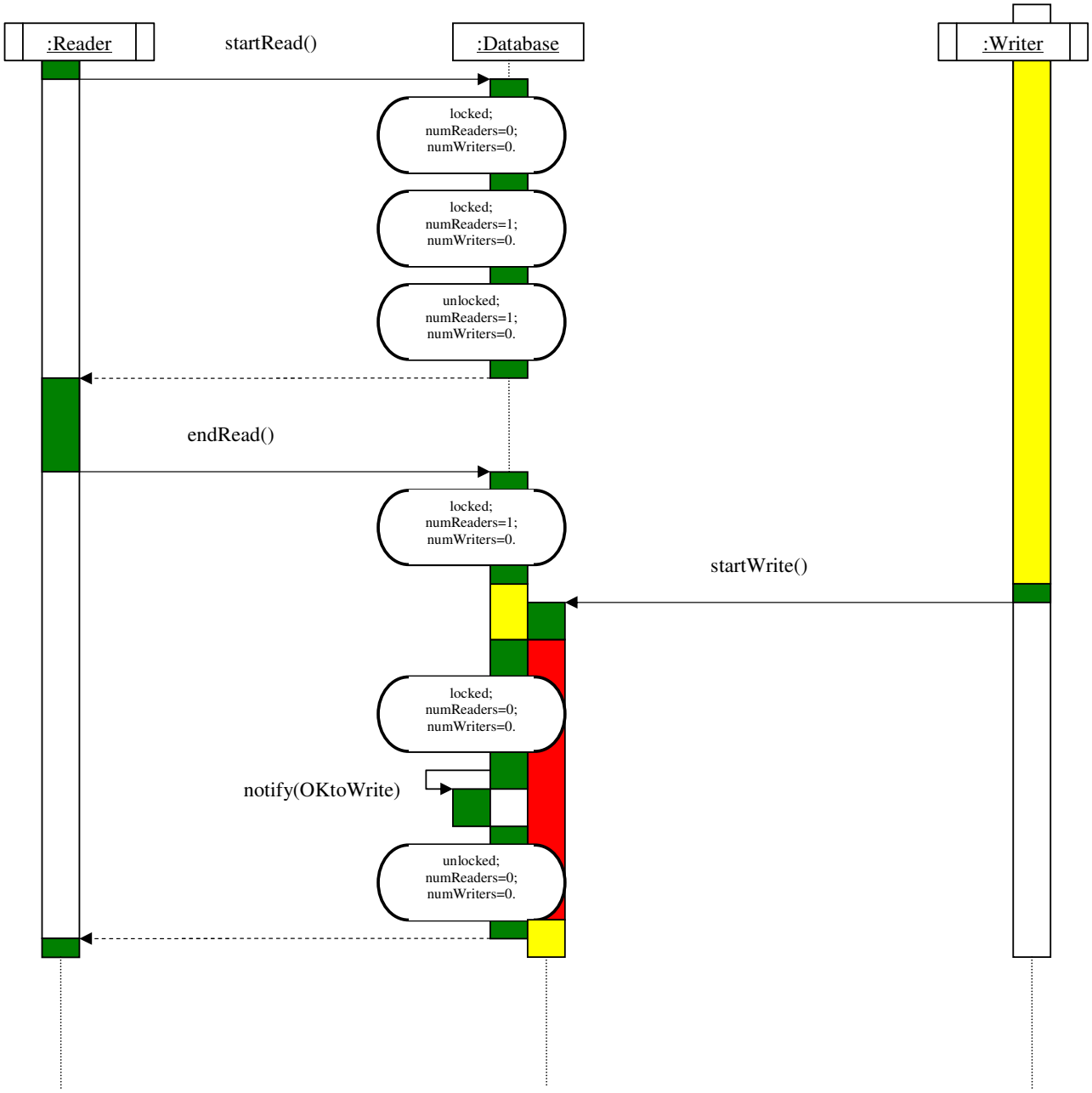
Scenario 5:


:Reader
startRead()
:Database
:Writer
locked;
numReaders=0;
numWriters=0.
locked;
numReaders=1;
numWriters=0.
startWrite()
unlocked;
numReaders=1;
numWriters=0.

Scenario 6


:Reader
:Database
startWrite()
:Writer
locked;
numReaders=0;
numWriters=0.
locked;
numReaders=0;
numWriters=1.
unlocked;
numReaders=0;
numWriters=1.
startRead()
locked;
numReaders=0;
numWriters=1.
wait(OKtoRead)
unlocked;
numReaders=0;
numWriters=1.

Scenario 7


:Reader
startRead()
:Database
:Writer
locked;
numReaders=0;
numWriters=0.
locked;
numReaders=1;
numWriters=0.
unlocked;
numReaders=1;
numWriters=0.
endRead()
locked;
numReaders=1;
numWriters=0.
startWrite()
locked;
numReaders=0;
numWriters=0.
notify(OKtoWrite)
unlocked;
numReaders=0;
numWriters=0.

Appendix C

Materials used in the two saUML vs. standard UML studies

C.1 saUML vs. standard UML experiment I: pre-test

Name: _____

I will participate in the experiment next Monday: **YES** **NO**

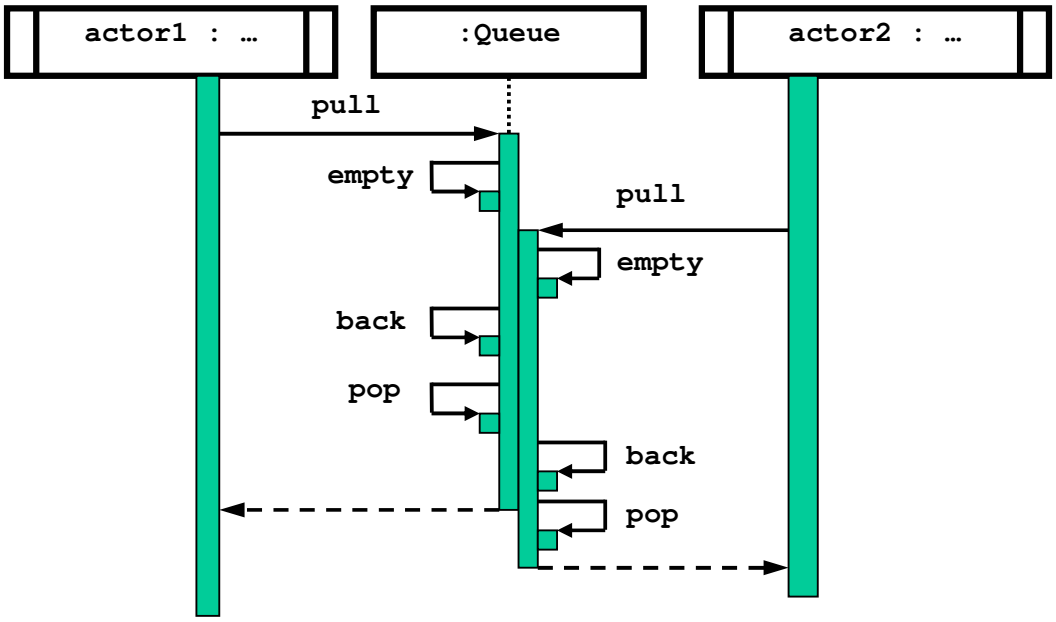


Diagram 1

_____ 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.
e. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* still contains Object A.

_____ Q2. Assume the *Queue* initially contains three objects, which were inserted in the order of A, B and C. 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 a copy of Object A; the *Queue* contains Object C only.
b. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* contains Object B and Object C.
c. actor1 gets a copy of Object A; actor2 gets a copy of Object B; the *Queue* contains Object C only.
d. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* becomes corrupted.
e. actor1 gets a copy of Object A; actor2 gets a copy of Object B; the *Queue* becomes corrupted.

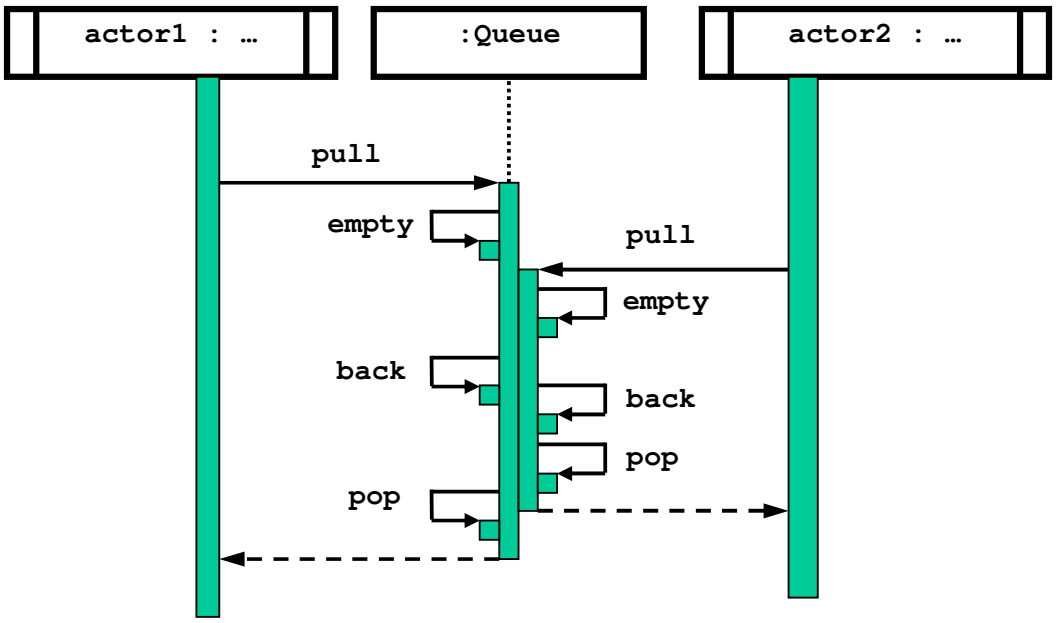


Diagram 2

_____ Q3. 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 2?

a. actor1 gets a copy of Object A; actor2 gets nothing; the *Queue* becomes empty.
b. actor1 gets nothing; actor2 gets a copy of Object A; the *Queue* becomes empty.
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 a copy of Object A; the *Queue* becomes corrupted.
e. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* still contains Object A.

_____ Q4. Assume the *Queue* initially contains three objects, which were inserted in the order of A, B and C. What happens as a result of actors 1 and 2 executing the pull method as seen in diagram 2?

a. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* contains Object C only.
b. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* contains Object B and Object C.
c. actor1 gets a copy of Object A; actor2 gets a copy of Object B; the *Queue* contains Object C only.
d. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* becomes corrupted.
e. actor1 gets a copy of Object A; actor2 gets a copy of Object B; the *Queue* becomes corrupted.

Note: In the following scenarios, the shared queue is implemented using monitors.

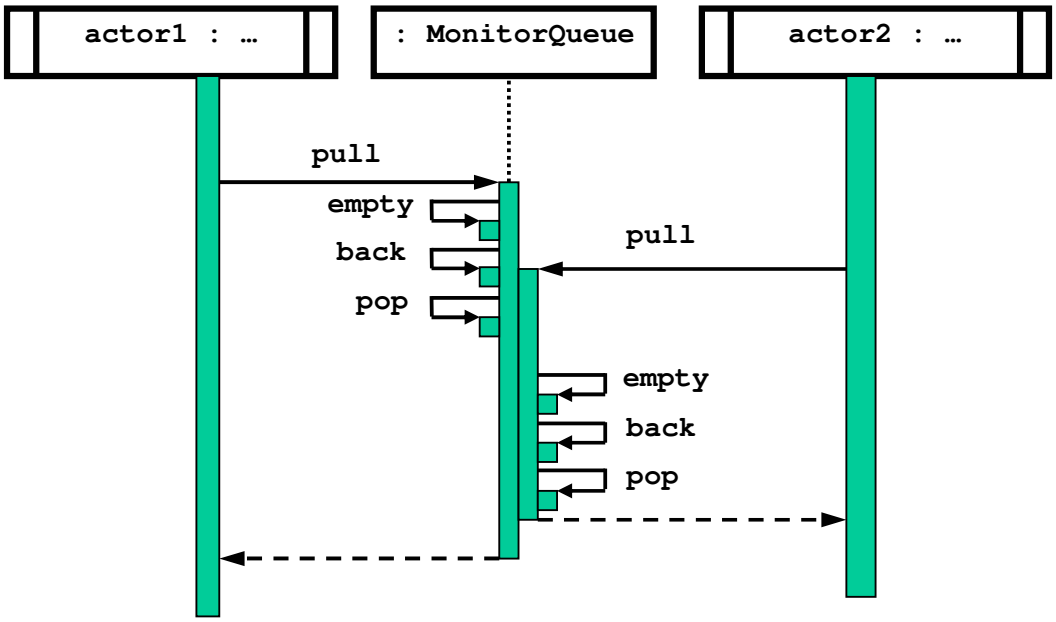


Diagram 3

_____ Q5. Assume the *Queue* initially contains three objects, which were inserted in the order A, B and C. Which of the following statements concerning diagram 3 is correct?

a. actor1 obtains the lock on the *MonitorQueue* after actor 2 obtains it.
b. actor1 releases the lock on the *MonitorQueue* after actor 2 executes *pop*.
c. The diagram is in error: actor 2 could not send the *pull* message to the monitor while actor1 is running in the *MonitorQueue*.
d. actor1 releases the lock then is context switched out and actor 2 is able to obtain the lock and complete its invocation of *pull* before actor1 is context switched in again.
e. The diagram is in error: actor1 must return from *pull* before actor 2 can proceed to execute *empty*, *back* and *pop*.

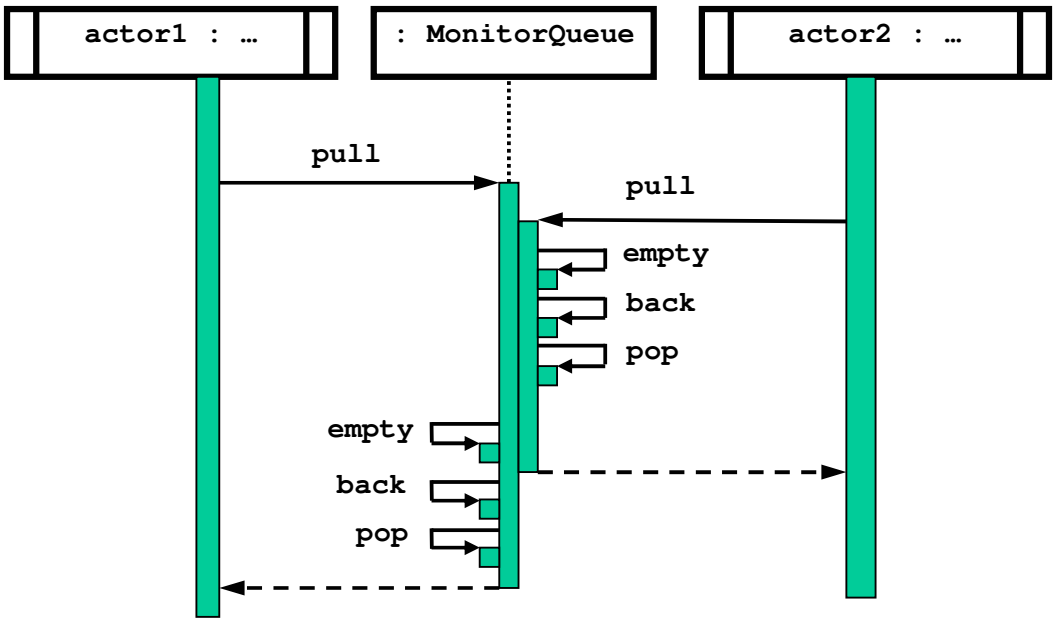


Diagram 4

_____ Q6. Assume the *Queue* initially contains three objects, which were inserted in the order A, B and C. Which of the following statements concerning diagram 4 is correct?

a. actor1 obtains the lock on the *MonitorQueue* after actor 2 obtains it.
b. actor2 releases the lock on the *MonitorQueue* after actor1 executes *empty*.
c. The diagram is in error: actor 2 could not send the *pull* message to the monitor while actor1 is running in the *MonitorQueue*.
d. actor2 releases the lock then is context switched out and actor1 is able to obtain the lock and complete its invocation of *pull* before actor 2 is context switched in again.
e. The diagram is in error: actor 2 must return from *pull* before actor1 can proceed to execute *empty*, *back* and *pop*.

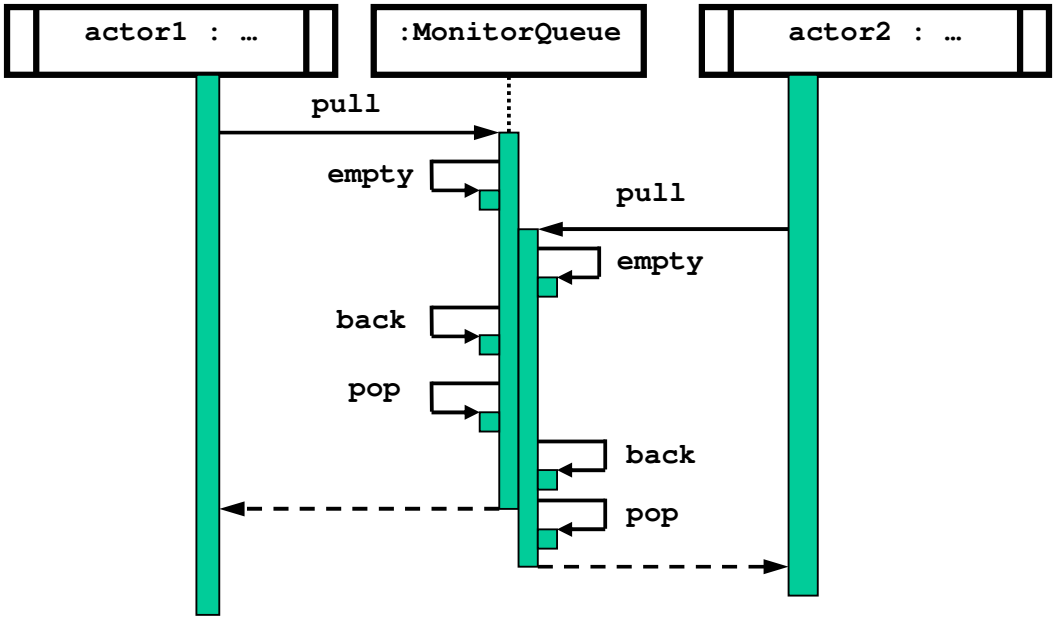


Diagram 5

_____ Q7. Assume the *MonitorQueue* initially contains three objects, which were inserted in the order A, B and C. Which of the following statements concerning diagram 5 is correct?

a. actor1 obtains the lock on the *MonitorQueue* after actor 2 obtains it.
b. actor1 releases the lock on the *MonitorQueue* after it executes *empty*.
c. The diagram is in error: actor 2 could not send the *pull* message to the monitor while actor1 is running in the *MonitorQueue*.
d. actor1 finishes *empty* then is context switched out and actor 2 is able to obtain the lock and proceed to execute *empty* before actor1 is context switched in again.
e. The diagram is in error: actor 2 could not proceed to execute *empty* while actor1 holds the lock on the *MonitorQueue*.

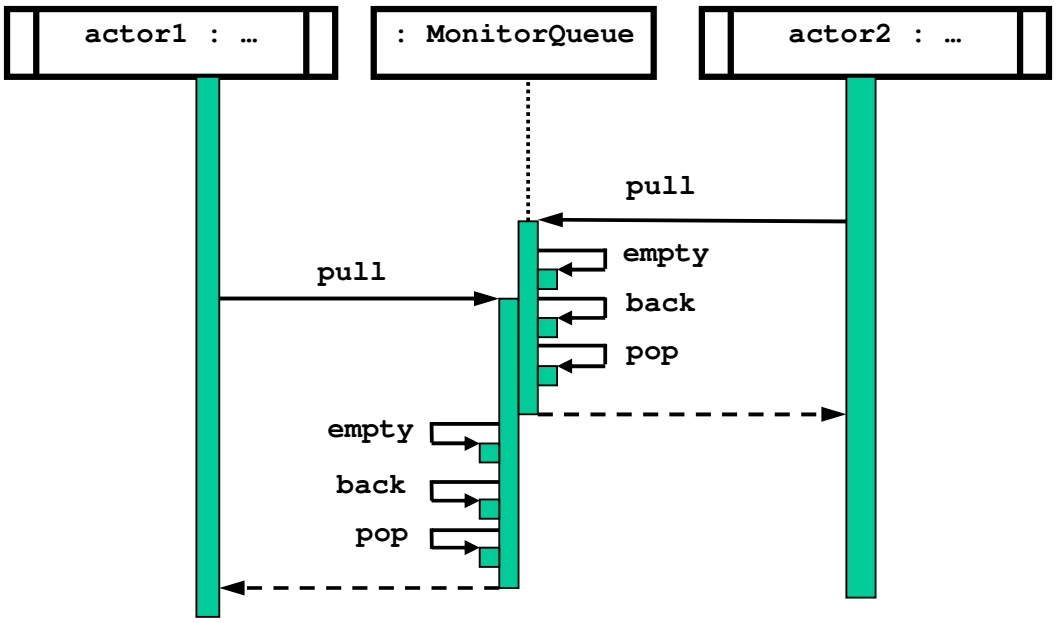


Diagram 6

_____ Q8. Assume the *Queue* initially contains three objects, which were inserted in the order A, B and C. What happens as a result of actors 1 and 2 executing the pull method as seen in diagram 6?

a. actor1 gets a copy of Object B; actor2 gets a copy of Object A; the *Queue* contains Object C only.
b. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* contains Object B and Object C.
c. actor1 gets a copy of Object A; actor2 gets a copy of Object B; the *Queue* contains Object C only.
d. actor1 gets a copy of Object B; actor2 gets a copy of Object A; the *Queue* becomes corrupted.
e. actor1 gets a copy of Object A; actor2 gets a copy of Object B; the *Queue* becomes corrupted.

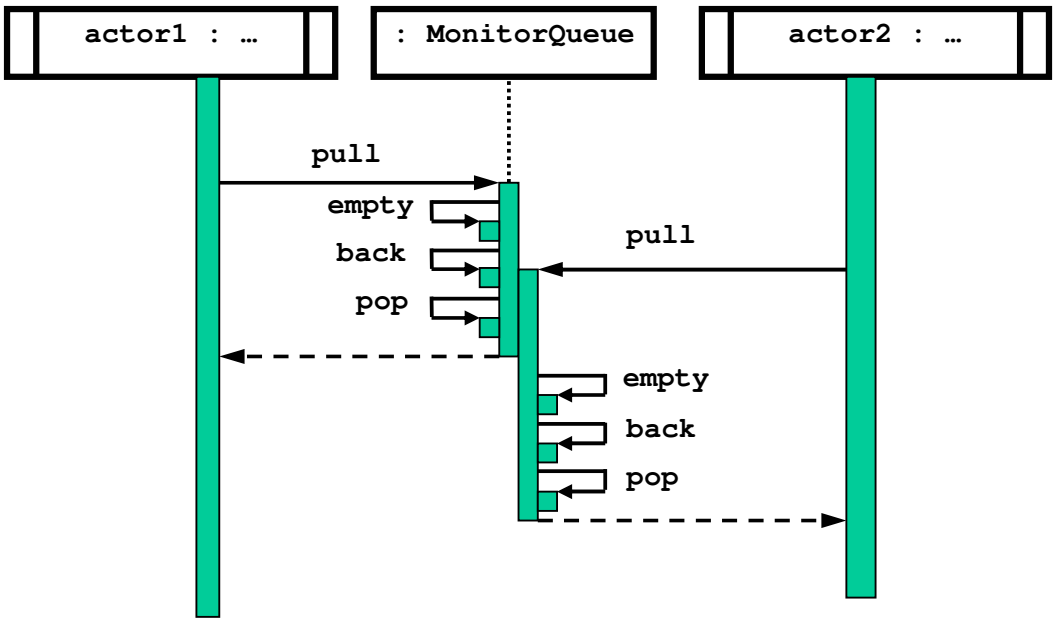


Diagram 7

_____ Q9. Why does actor 2 delay after invoking the pull method and before invoking the empty method as seen in diagram 7?

a. actor1 is in the monitor; thus no other thread may execute until actor1 leaves the monitor.
b. actor2 was delayed because the thread running within actor1 was switched in to execute just prior to actor2's invocation of *empty*. Had this not happened, actor 2 could proceed to execute *empty* before actor1 completes its invocation of the *pull* method.
c. The queue is empty and actor2 must wait until a new object is placed in the queue.
d. The diagram is in error: actor2 could not send the *pull* message while actor1 is running in the monitor.
e. Because actor1 holds the lock on the monitor, actor2 must wait for actor1 to release the lock.

Q10. Briefly explain what can go wrong when multiple actors access shared data without synchronization.

_____ Q11. Which of the following does a monitor guarantee?

a. Only one thread at a time may execute within the monitor
b. Once a thread *T* enters the monitor, no other thread may execute within any object until *T* leaves the monitor
c. Multiple threads may enter the monitor at the same time provided that none of these threads modifies any of the data in the monitor.
d. All of the above
e. None of the above

C.2 saUML vs. standard UML experiment I: post-test

C.2.1 Experiment I: post-test questions

Name: _____

The following questions refer to the "Scenario" diagrams that you have received in a separate packet. In answering the questions below, you should be sure to note which scenario the question is referring to. Also, you may assume the following:

- **the initial Balance stored in the Database for our accountID of interest is $180**
- **the accountID attribute is not in these scenarios; thus client1 and client2 will always be referring to the same account**
- **the conversion rate is $1.80 per £ 1 (GBP). Thus:**

USD	GBP
$ 90	£ 50
$ 180	£ 100
$ 270	£ 150
$ 360	£ 200

Questions 1 and 2 refer to Scenario 1.

1. What happens as a result of the execution depicted in Scenario 1?
    a. *withdrawHalf* returns **£**50 and the account now contains $90
    b. *withdrawHalf* returns **£**100 and the account now contains $180
    c. *withdrawHalf* returns **£**50 and the account now contains $180
    d. *withdrawHalf* returns **£**100 and the account now contains $90

2. The *withdrawHalf* activation experiences a delay between its invocations of *getBalance* and *setBalance*, because:
    a. the client2 thread must wait to obtain the lock for the IBA object
    b. the client2 thread was context-switched out while the client1 thread was context-switched in and executed
    c. the client2 thread must wait for the *deposit* activation to complete before it may modify the database
    d. none of the above

Question 3 refers to Scenario 2.

3. What happens as a result of the execution depicted in Scenario 2?
    a. *withdrawHalf* returns **£**50 and the account now contains $90
    b. *withdrawHalf* returns **£**100 and the account now contains $180
    c. *withdrawHalf* returns **£**50 and the account now contains $180
    d. *withdrawHalf* returns **£**100 and the account now contains $90

Note: Scenarios 3 through 7 use the MonitorIBA class rather than the IBA class.

Questions 4 and 5 refer to Scenario 3.

4. What happens as a result of the execution depicted in Scenario 3?
    a. *withdrawHalf* returns **£**50 and the account now contains $90
    b. *withdrawHalf* returns **£**100 and the account now contains $180
    c. the diagram is in error; the client2 thread must wait for the client1 thread to release the lock before it can proceed
    d. the diagram is in error; the client2 thread cannot invoke the *withdrawHalf* method until after the client1 thread returns from its invocation of *deposit*

5. What must have happened after the deposit activation invokes *setBalance* and before the *withdrawHalf* activation invokes *getBalance*?
    a. a context-switch occurs: the client1 thread switches out, and the client2 thread switches in
    b. the client1 thread releases the lock
    c. the client2 thread acquires the lock
    d. all of the above

Question 6 refers to Scenario 4.

6. Which of the following statements concerning the diagram for Scenario 4 is correct?
    a. *withdrawHalf* returns **£**50 and the account now contains $90
    b. *withdrawHalf* returns **£**100 and the account now contains $180
    c. the diagram is in error; the client2 thread must wait for the client1 thread to release the lock before it can proceed
    d. the diagram is in error; the client2 thread cannot invoke the *withdrawHalf* method until after the client1 thread returns from its invocation of *deposit*

Question 7 refers to Scenario 5.

7. Which of the following statements concerning the diagram for Scenario 5 is correct?
    a. *withdrawHalf* returns **£**50 and the account now contains $270
    b. *withdrawHalf* returns **£**100 and the account now contains $180
    c. the diagram is in error; the client1 thread must wait for the client2 thread to release the lock before it can proceed
    d. the diagram is in error; the client1 thread cannot invoke the *deposit* method until after the client2 thread returns from *withdrawHalf*

Question 8 refers to Scenario 6.

8. What happens as a result of the execution depicted in Scenario 6?
    a. *withdrawHalf* returns **£**50 and the account now contains $90
    b. *withdrawHalf* returns **£**100 and the account now contains $270
    c. *withdrawHalf* returns **£**50 and the account now contains $270
    d. *withdrawHalf* returns **£**100 and the account now contains $180

Question 9 refers to Scenario 7.

9. Why does the *withdrawHalf* activation experience a delay before invoking the *getBalance* method as seen in Scenario 7?

    a. While a monitor permits multiple simultaneous activations, it allows only one of them to execute, and this executing activation must execute to completion before any of the other activations will be allowed to execute. Thus the *withdrawHalf* activation cannot execute until the *deposit* activation completes by returning to client1.
    b. The activation was delayed because the client1 thread was switched in to execute just prior to the *withdrawHalf* activation's invocation of *getBalance*. Had this not happened, the *withdrawHalf* activation could have proceeded to invoke *getBalance* before the *deposit* activation invoked *setBalance*.
    c. The diagram is in error: A monitor does not permit simultaneous activations; thus client2 could not have initiated the *withdrawHalf* activation until after the *deposit* activation completed.
    d. Because the client1 thread holds the lock on the monitor, the client2 thread must wait for the client1 thread to release the lock.

Questions 10 – 14 do not refer to any particular Scenario.

10. When multiple clients access a shared database (either directly or indirectly) such as may happen given the example code for class IBA presented here,
    a. a deposit can be lost, and the client "loses" money
    b. a withdrawal can be lost, and the client "gains" money
    c. both a and b
    d. neither a nor b

11. When multiple clients access a shared database (either directly or indirectly) such as may happen given the example code for class IBA presented here,
    a. a deposit can be double-counted, and the client "gains" money
    b. a withdrawal can be double-counted, and the client "loses" money
    c. both a and b
    d. neither a nor b

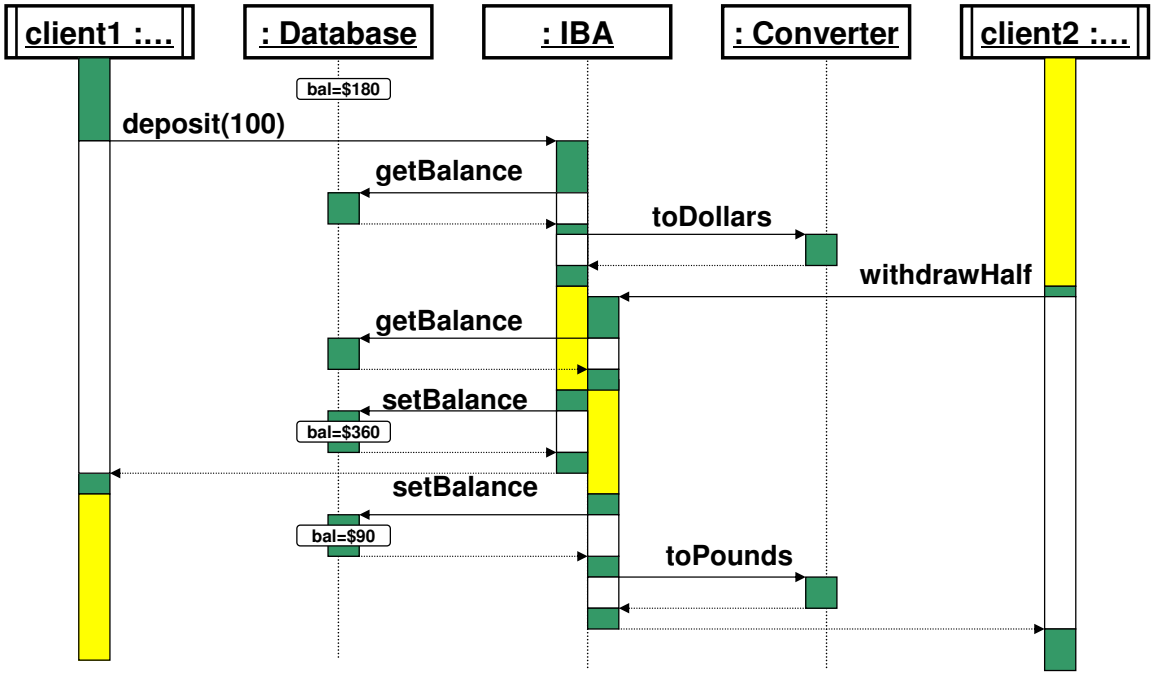
12. Using the monitor version of the bank account, as in the example code presented here:
    a. ensures that when multiple threads concurrently activate methods on the MonitorIBA object, the result of the completion of these multiple activations is some non-interleaved sequence of complete executions of these methods
    b. ensures that multiple concurrent activations of methods on the account object are performed in the order in which they are initiated
    c. gives priority to *deposit* activations over *withdrawHalf* activations
    d. none of the above

13. Context-switching, in which the running thread is interrupted and another thread begins to run
    a. happens only in the case of unprotected shared data structures
    b. happens only in the case of protected shared data structures, such as monitors
    c. is prevented in the case of protected shared data structures, such as monitors
    d. none of the above

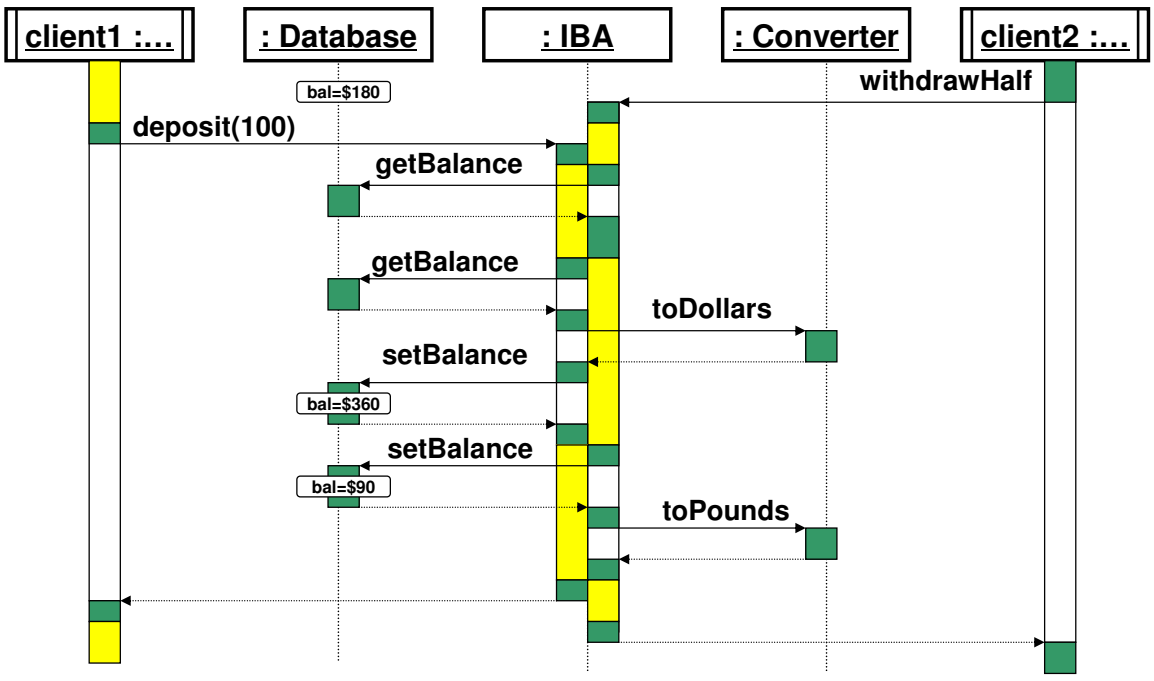
14. The purpose of the lock that is associated with a monitor is to:
    a. serialize the execution of monitor methods
    b. ensure that monitor methods are invoked in some specified order
    c. to ensure fairness through context-switching
    d. none of the above

C.2.2 Experiment I: post-test saUML sequence diagrams

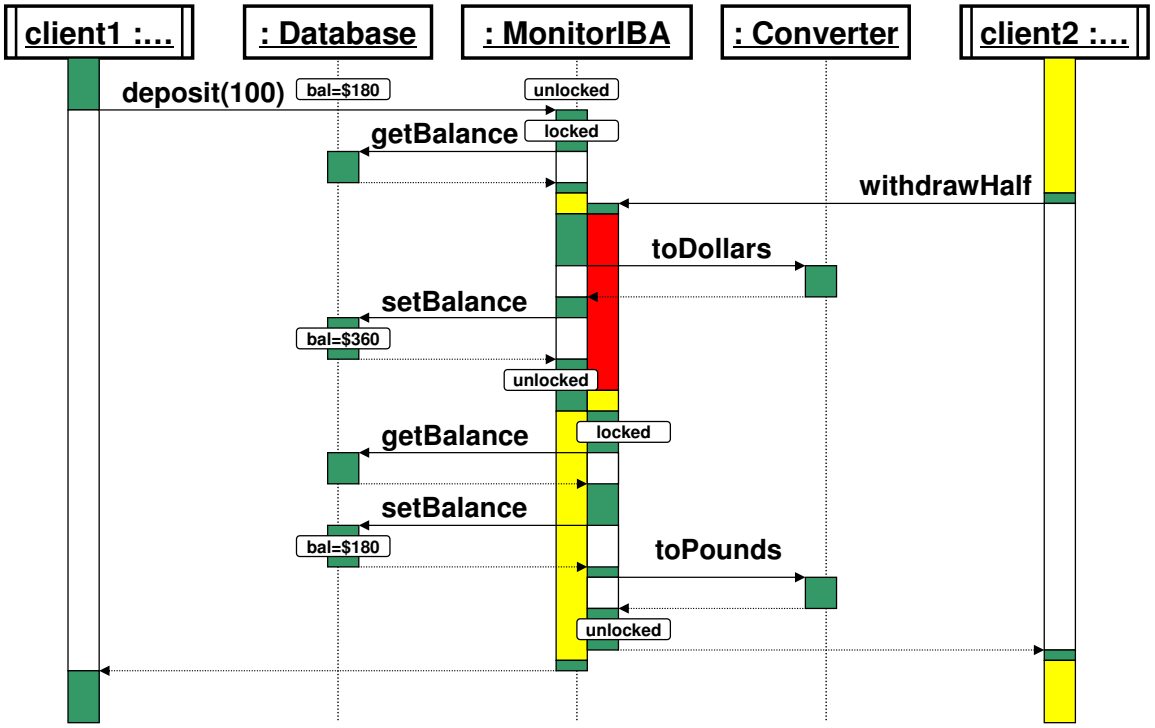
Scenario 1



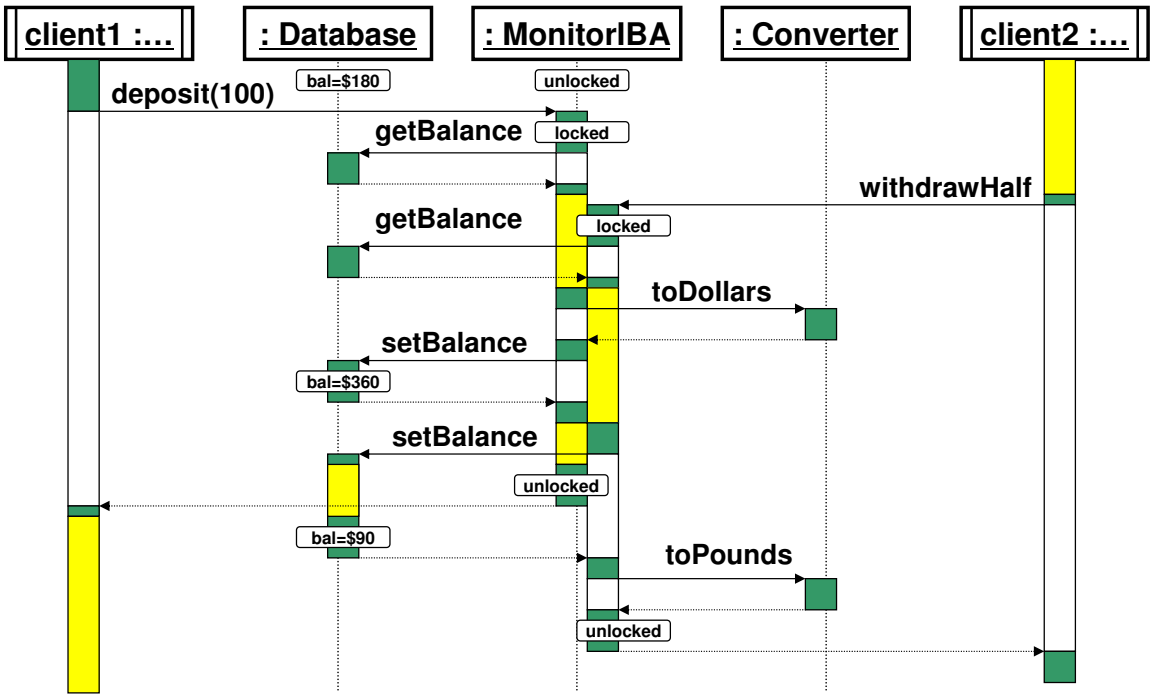
Scenario 2



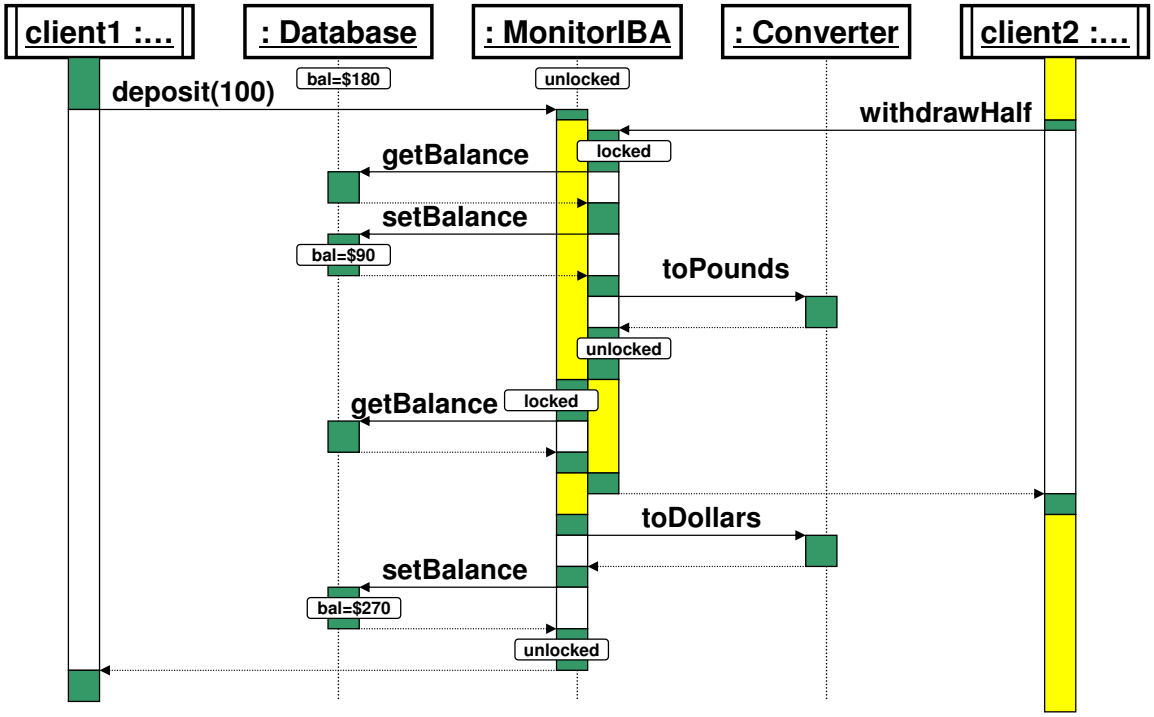
Scenario 3



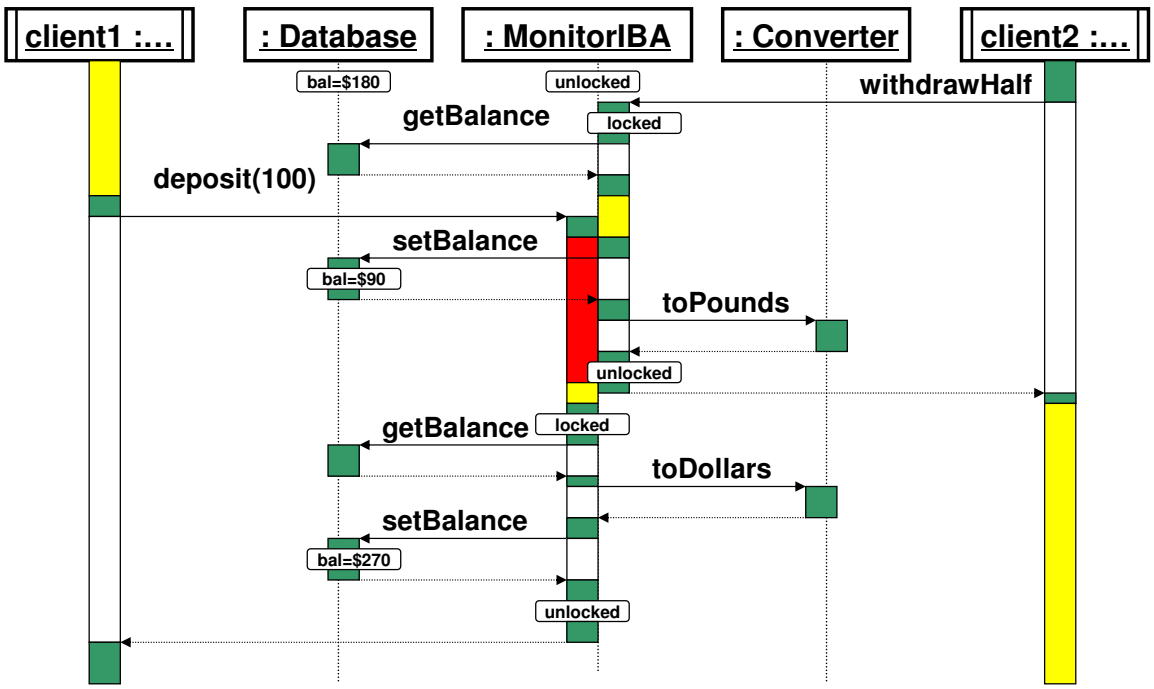
Scenario 4



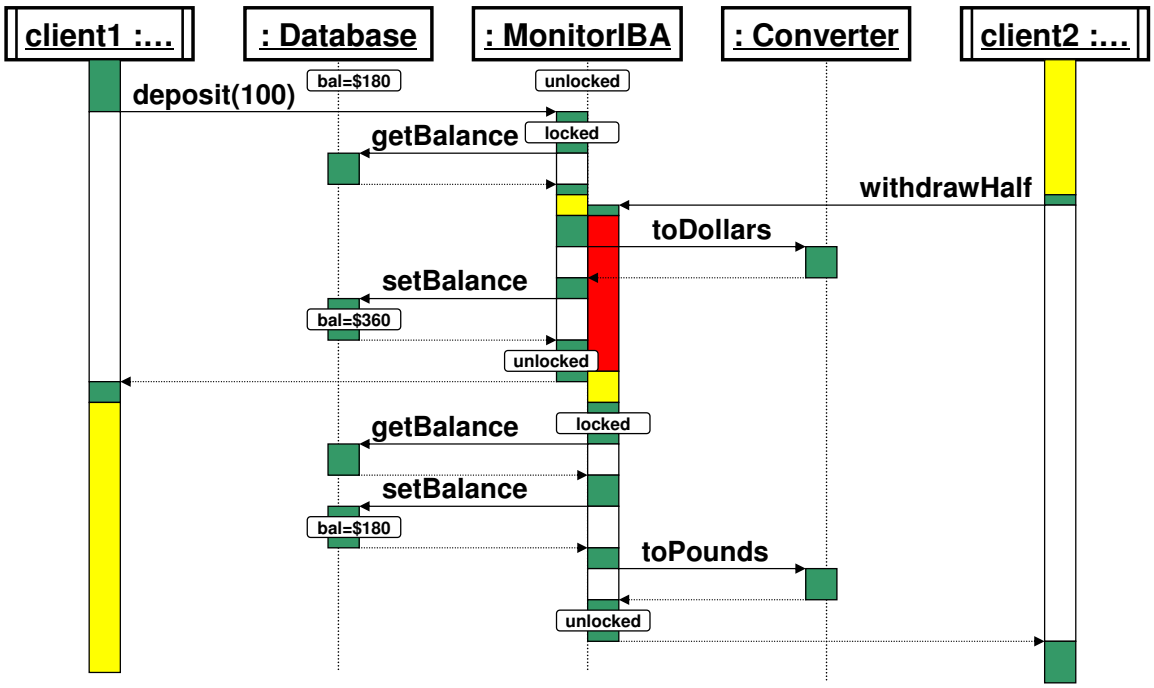
Scenario 5



Scenario 6

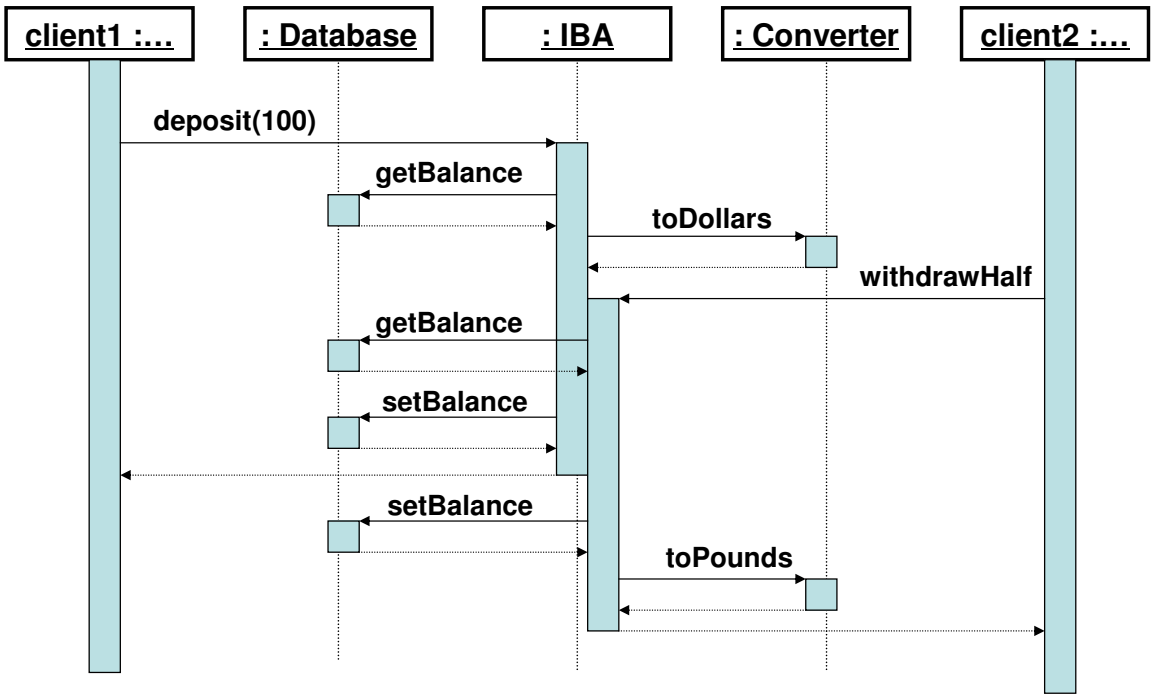


Scenario 7

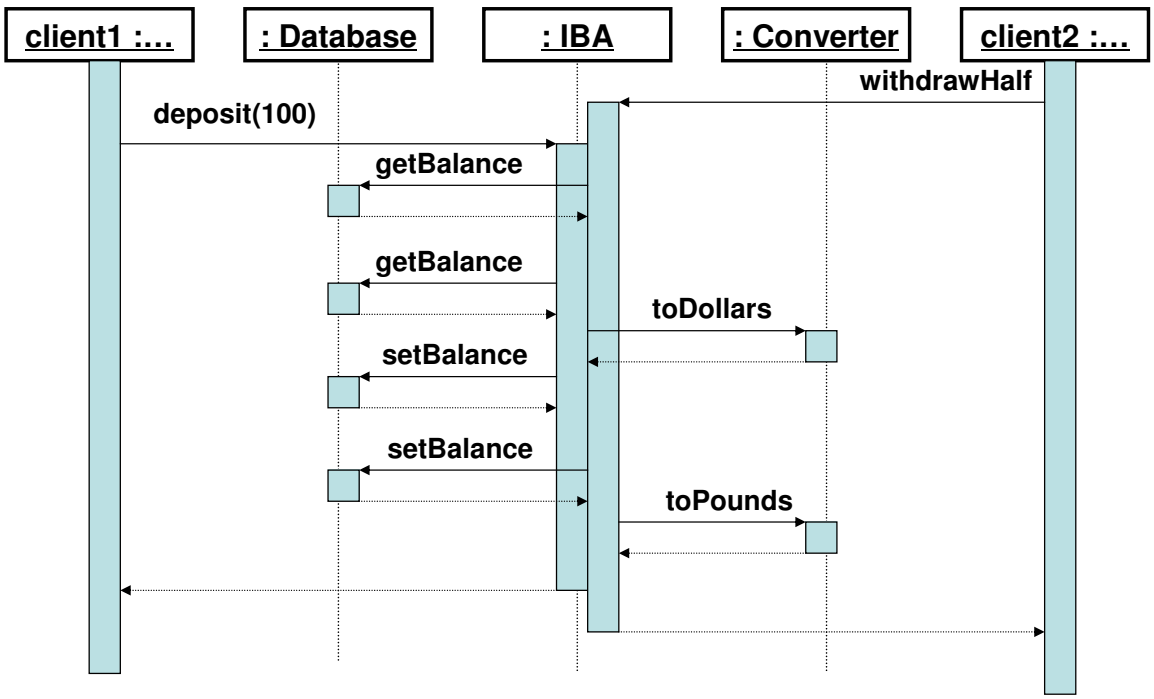

client1 :...
: Database
: MonitorIBA
: Converter
client2 :...
bal=$180
unlocked
deposit(100)
getBalance
locked
withdrawHalf
toDollars
setBalance
bal=$360
unlocked
getBalance
locked
setBalance
bal=$180
toPounds
unlocked

C.2.3 Experiment I: post-test UML sequence diagrams

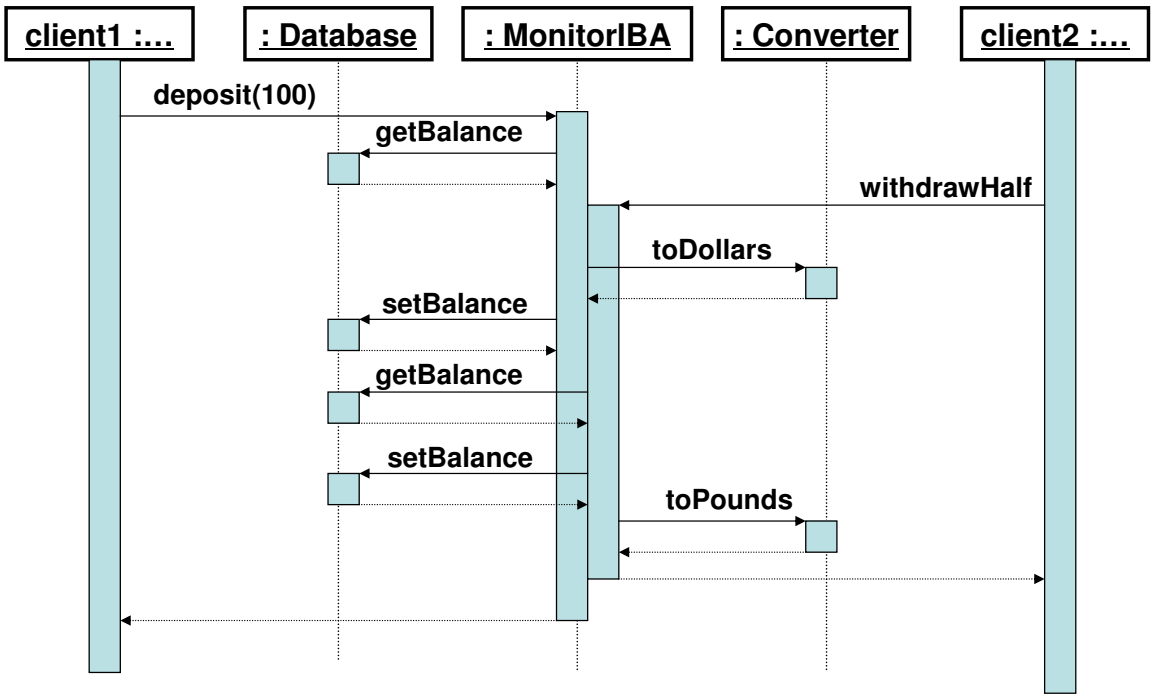
Scenario 1



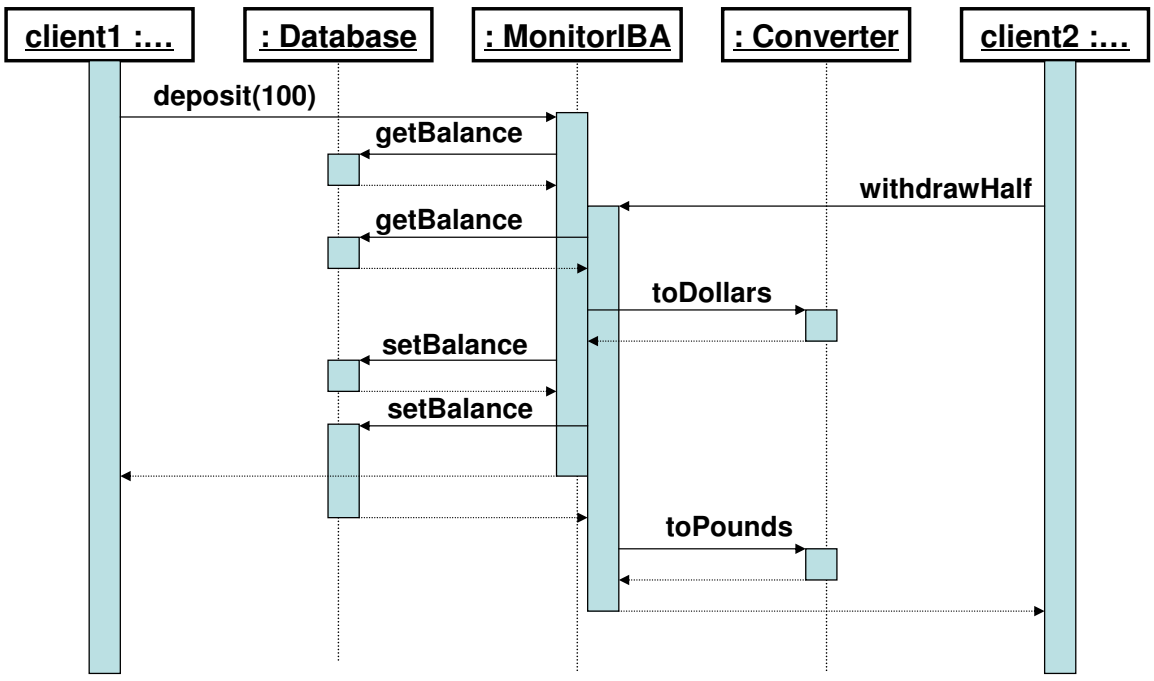
Scenario 2



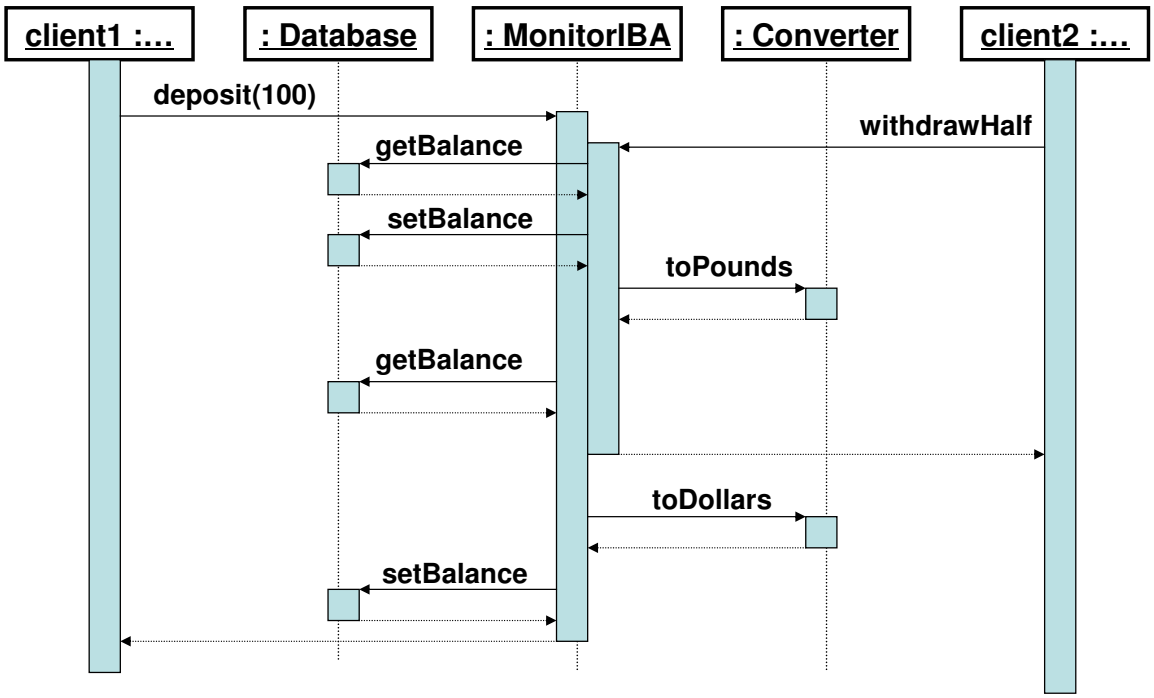
Scenario 3



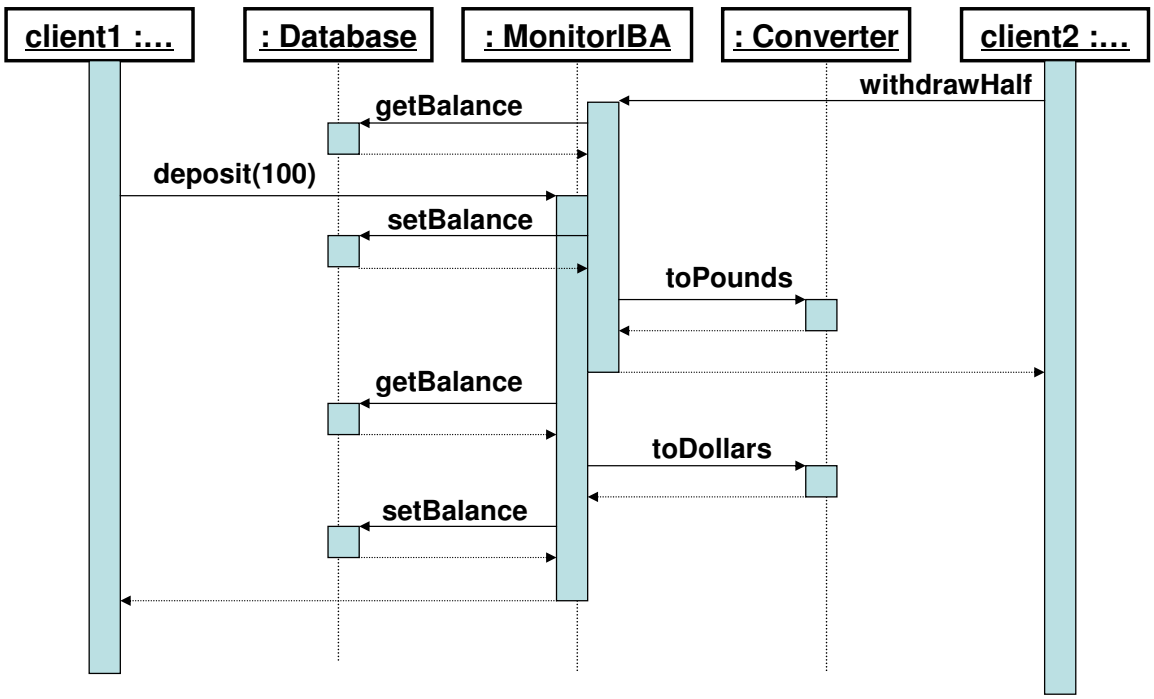
Scenario 4



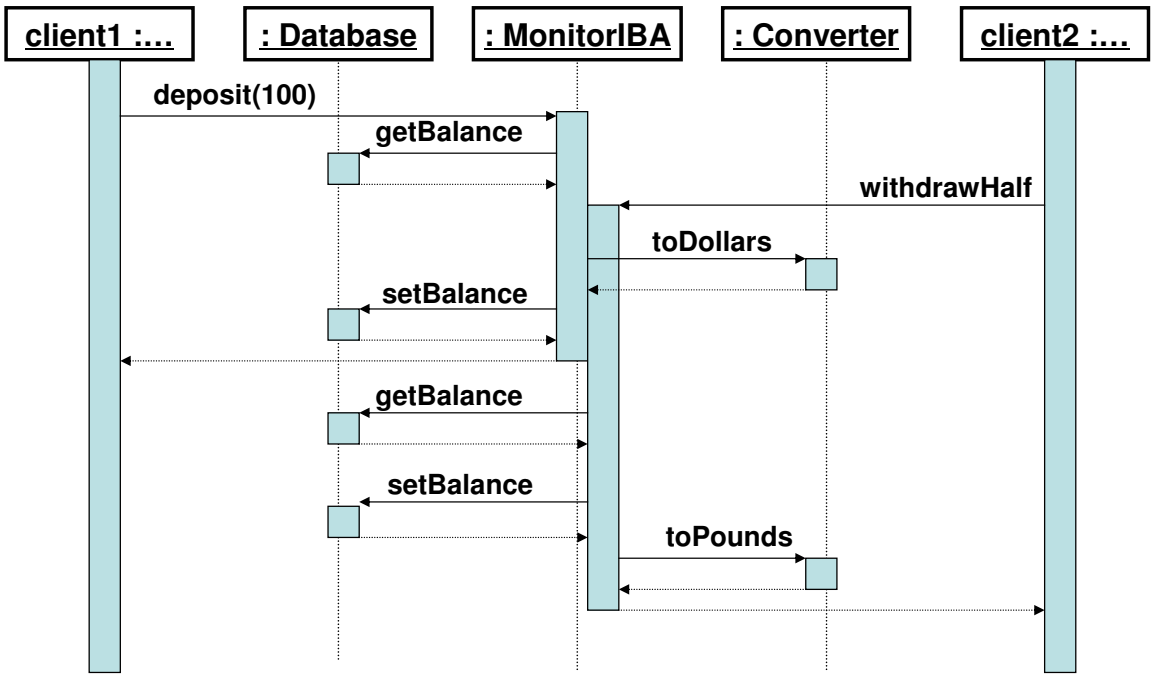
Scenario 5



Scenario 6



Scenario 7


client1 :...
: Database
: MonitorIBA
: Converter
client2 :...
deposit(100)
getBalance
withdrawHalf
toDollars
setBalance
getBalance
setBalance
toPounds

C.3 saUML vs. standard UML experiment II: pre-test

CSE 335 Extra Credit Option 1

Part I: Pre-test in conjunction with user study

Name: _____

Questions 1 and 2 below refer to a system comprising two active objects—`actor1` and `actor2`—each hosted by a dedicated thread of control. **Figure 1** depicts an interaction in which these actors attempt (concurrently) to pull an element off a `Queue` object, which implements its `pull` operation by invoking a sequence of operations on another object `q` of type `deque<string>`. Please note that the `Queue` object is not implemented according to the monitor-object pattern.

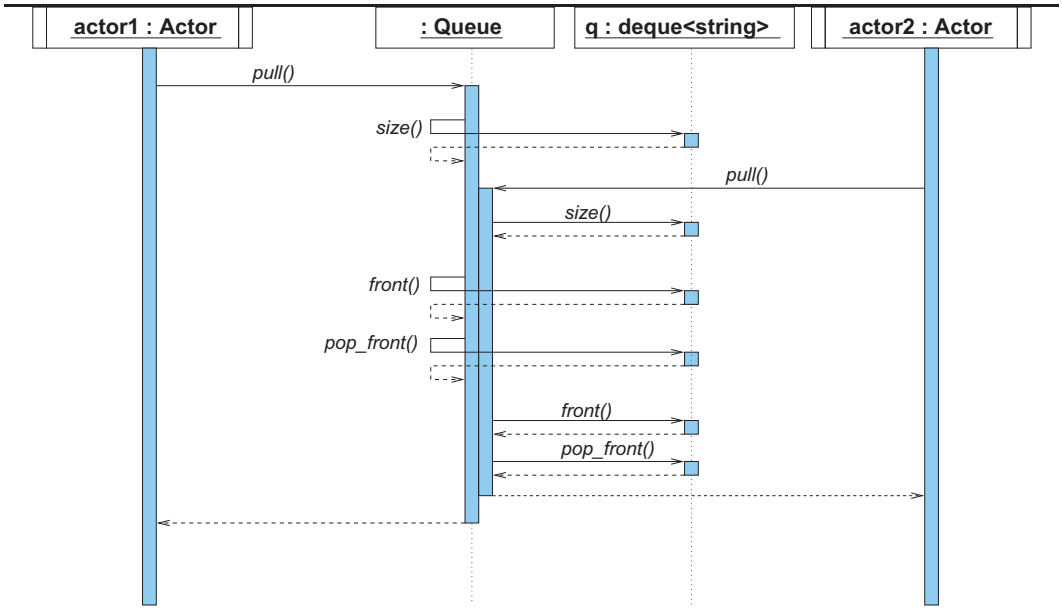


Figure 1: Two actors concurrently accessing an unprotected queue.

1. Assume, at the beginning of the interaction depicted in **Figure 1**, that the queue (`q`) contains only the string "a". What happens as a result of this interaction?

   (a) `actor1` gets a copy of string "a"; `actor2` gets nothing; `q` becomes empty.

   (b) `actor1` gets a copy of string "a"; `actor2` gets nothing; `q` becomes corrupted.

   (c) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "a"; `q` becomes corrupted.

   (d) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "a"; `q` becomes empty.

   (e) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "a"; `q` contains the string "a" at the end of the interaction.

2. Assume, at the beginning of the interaction depicted in **Figure 1**, that the queue (`q`) contains the strings "a", "b", and "c" inserted in that order. What happens as a result of this interaction?

   (a) `actor1` gets a copy of "a"; `actor2` gets a copy of "a"; `q` contains the string "c" only.

   (b) `actor1` gets a copy of "a"; `actor2` gets a copy of "a"; `q` contains the strings "b" and "c".

   (c) `actor1` gets a copy of "a"; `actor2` gets a copy of "b"; `q` contains the string "c" only.

   (d) `actor1` gets a copy of "a"; `actor2` gets a copy of "a"; `q` becomes corrupted.

   (e) `actor1` gets a copy of "a"; `actor2` gets a copy of "b"; `q` becomes corrupted.

As with Questions 1 and 2, Questions 3 and 4 refer to a system comprising two active objects—`actor1` and `actor2`—each hosted by a dedicated thread of control. **Figure 2** depicts another interaction in which these actors attempt (concurrently) to pull an element off a `Queue` object, which implements its `pull` operation by invoking a sequence of operations on another object `q` of type `deque<string>`. Again, the `Queue` object is not implemented according to the monitor-object pattern.

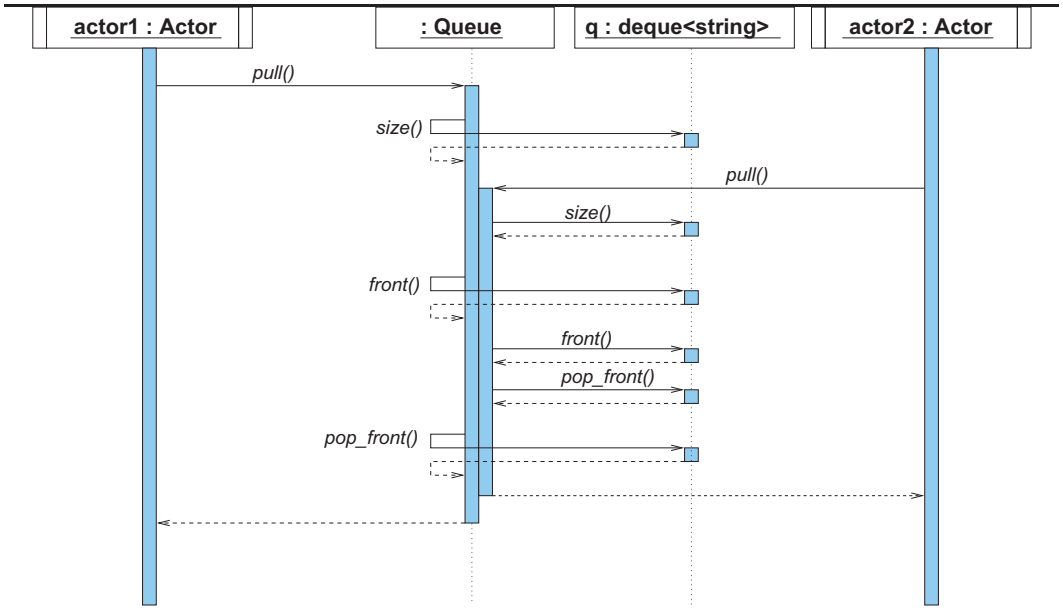


Figure 2: Two actors concurrently accessing an unprotected queue

3. Assume, at the beginning of the interaction depicted in **Figure 2**, that the queue (`q`) contains only the string "a". What happens as a result of this interaction?

   (a) `actor1` gets a copy of string "a"; `actor2` gets nothing; `q` becomes empty.

   (b) `actor1` gets nothing; `actor2` gets a copy of string "a"; `q` becomes empty.

   (c) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "a"; `q` becomes empty.

   (d) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "a"; `q` becomes corrupted.

   (e) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "a"; `q` still contains string "a".

4. Assume, at the beginning of the interaction depicted in **Figure 2**, that the queue (`q`) contains the strings "a", "b", and "c" inserted in that order. What happens as a result of this interaction?

   (a) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "a"; `q` contains string "c" only.

   (b) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "a"; `q` contains strings "b" and "c" only.

   (c) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "b"; `q` contains string "c" only.

   (d) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "a"; `q` becomes corrupted.

   (e) `actor1` gets a copy of string "a"; `actor2` gets a copy of string "b"; `q` becomes corrupted.

Question 5 below refers to a system comprising two active objects—`actor1` and `actor2`—each hosted by a dedicated thread of control. **Figure 3** depicts an interaction in which these actors attempt (concurrently) to pull an element off a `MonitorQueue` object, which implements its `pull` operation invoking a sequence of operations on another object `q` of type `deque<string>`. As its name suggests, the `MonitorQueue` object is implemented according to the monitor-object pattern.

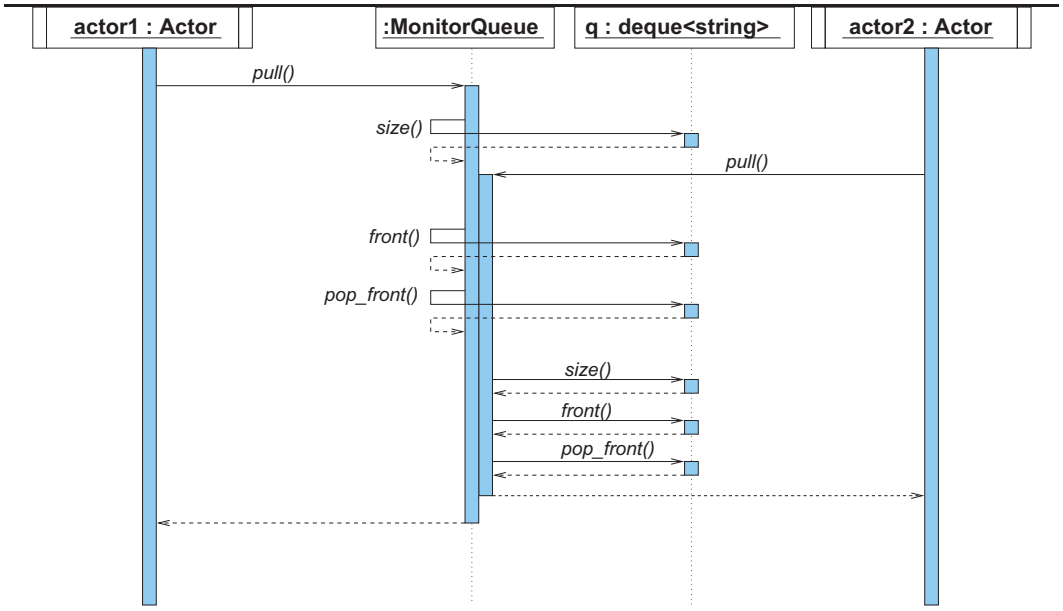


Figure 3: Two actors accessing a queue implemented as a monitor

5. Assume, at the beginning of the interaction depicted in **Figure 3**, that the queue (`q`) contains three elements "a", "b", and "c" inserted in that order. Which of the following statements is correct?

   (a) `actor1` obtains the lock on the `MonitorQueue` after `actor2` obtains it.

   (b) `actor1` releases the lock on the `MonitorQueue` after the thread hosting `actor2` executes `pop_front`.

   (c) The diagram is in error; `actor2` could not send the `pull` message to the monitor while `actor1` is running in the `MonitorQueue`.

   (d) `actor1` releases the lock and is then context switched out, after which `actor2` is able to obtain the lock and complete its invocation of `pull` before `actor1` is context switched in again.

   (e) The diagram is in error; actorOne must return from `pull` before `actor2` may proceed to execute `size`, `front`, and `pop_front`.

Question 6 below refers to a system comprising two active objects—`actor1` and `actor2`—each hosted by a dedicated thread of control. **Figure 4** depicts an interaction in which these actors attempt (concurrently) to pull an element off a `MonitorQueue` object, which implements its `pull` operation invoking a sequence of operations on another object `q` of type `deque<string>`. As its name suggests, the `MonitorQueue` object is implemented according to the monitor-object pattern.

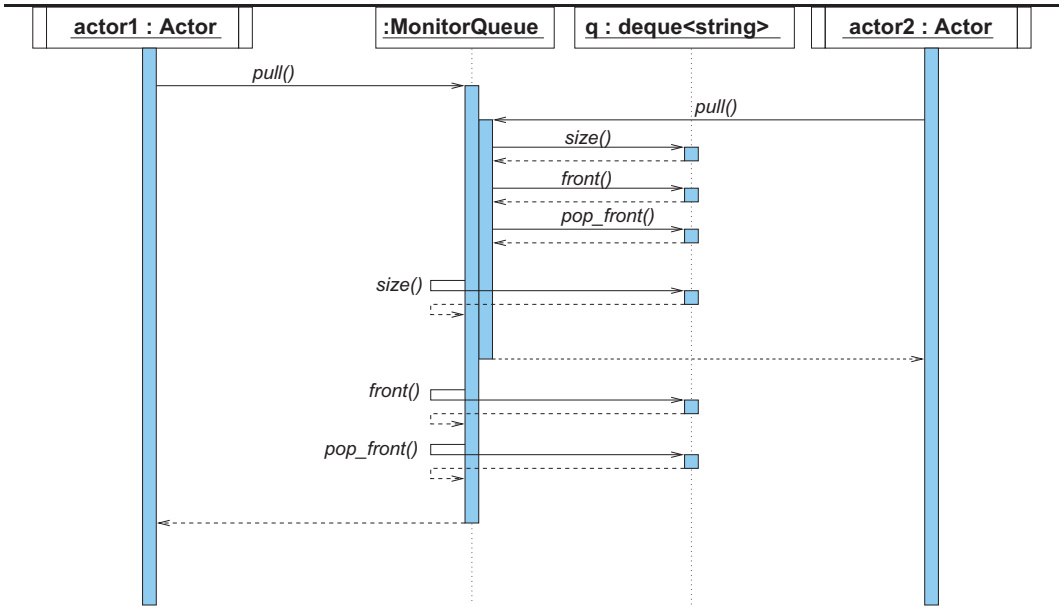


Figure 4: Two actors accessing a queue implemented as a monitor

6. Assume, at the beginning of the interaction depicted in **Figure 4**, that the queue (`q`) contains three elements "a", "b", and "c" inserted in that order. Which of the following statements is correct?

   (a) `actor1` obtains the lock on the `MonitorQueue` after `actor2` obtains it.

   (b) `actor2` releases the lock on `MonitorQueue` after `actor1` executes `size`.

   (c) The diagram is in error: `actor2` could not send the `pull` message to the `MonitorQueue` while `actor1` is activated within it.

   (d) `actor2` releases the lock then is context switched out and `actor1` is able to obtain the lock and complete its invocation of `pull` before `actor2` is context switched in again.

   (e) The diagram is in error: `actor2` must return from `pull` before `actor1` may proceed to execute `size`, `front`, and `pop_front`.

Question 7 below refers to a system comprising two active objects—`actor1` and `actor2`—each hosted by a dedicated thread of control. **Figure 5** depicts an interaction in which these actors attempt (concurrently) to pull an element off a `MonitorQueue` object, which implements its `pull` operation invoking a sequence of operations on another object `q` of type `deque<string>`. As its name suggests, the `MonitorQueue` object is implemented according to the monitor-object pattern.

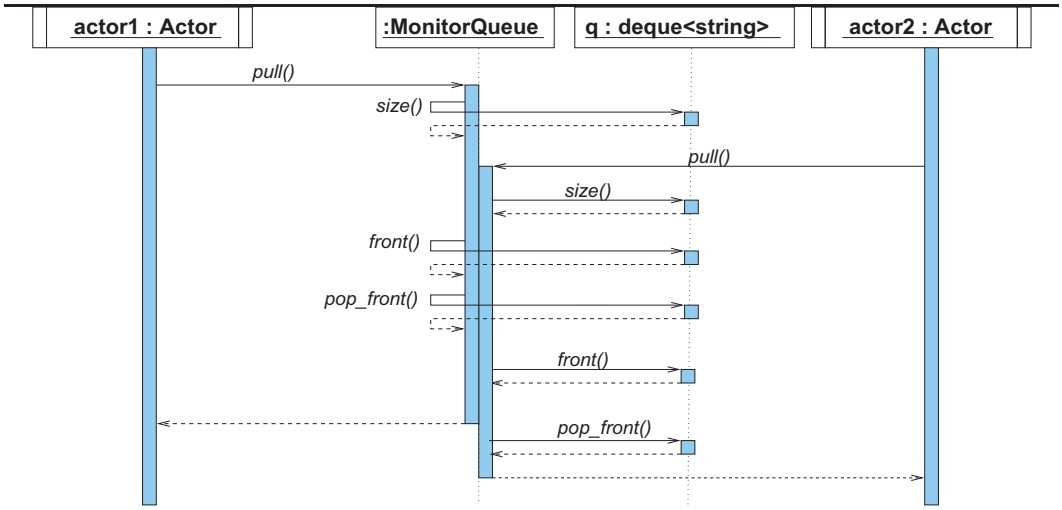


Figure 5: Two actors accessing a queue implemented as a monitor

7. Assume, at the beginning of the interaction depicted in **Figure 5**, that the queue (`q`) contains three elements "a", "b", and "c" inserted in that order. Which of the following statements is correct?

   (a) `actor1` obtains the lock on the `MonitorQueue` after `actor2` obtains it.

   (b) `actor1` releases the lock on the `MonitorQueue` after it executes `size`.

   (c) The diagram is in error: `actor2` could not send the `pull` message to the monitor while `actor1` is running inside the `MonitorQueue`.

   (d) `actor1` finishes `size` then is context switched out and `actor2` is able to obtain the lock and proceed to execute `size` before `actor1` is context switched in again.

   (e) The diagram is in error: `actor2` could not proceed to execute `size` while `actor1` holds the lock on `MonitorQueue`.

Question 8 below refers to a system comprising two active objects—`actor1` and `actor2`—each hosted by a dedicated thread of control. **Figure 6** depicts an interaction in which these actors attempt (concurrently) to pull an element off a `MonitorQueue` object, which implements its `pull` operation invoking a sequence of operations on another object `q` of type `deque<string>`. As its name suggests, the `MonitorQueue` object is implemented according to the monitor-object pattern.

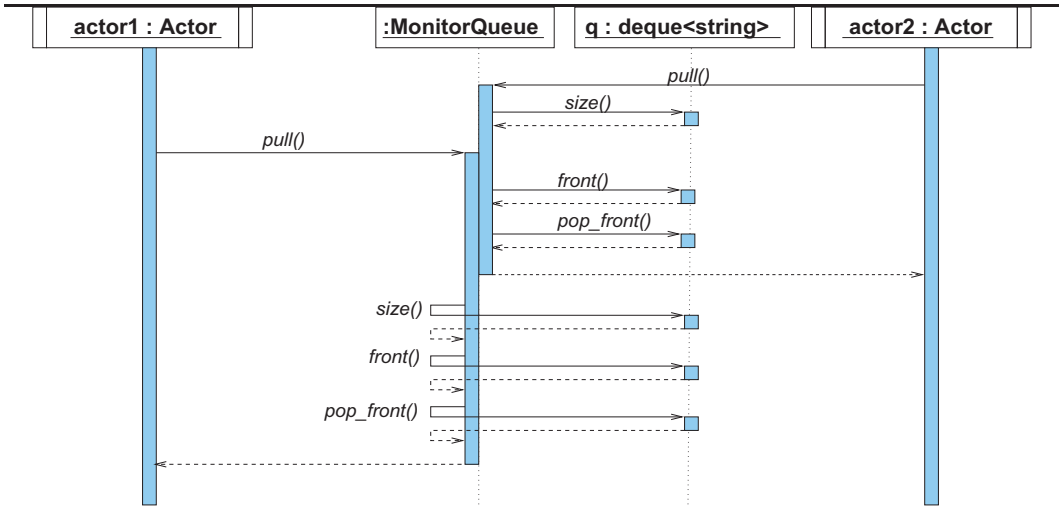


Figure 6: Two actors accessing a queue implemented as a monitor

8. Assume, at the beginning of the interaction depicted in **Figure 6**, that the queue (`q`) contains three elements “a”, “b”, and “c” inserted in that order. Which of the following statements is correct?

   (a) `actor1` gets a copy of string “b”; `actor2` gets a copy of string “a”; `q` contains string “c” only.

   (b) `actor1` gets a copy of string “a”; `actor2` gets a copy of string “a”; `q` contains strings “b” and “c”.

   (c) `actor1` gets a copy of string “a”; `actor2` gets a copy of string ”b”; `q` contains string “c” only.

   (d) `actor1` gets a copy of string “b”; `actor2` gets a copy of string “a”; `q` becomes corrupted.

   (e) `actor1` gets a copy of string “a”; `actor2` gets a copy of string ”b”; `q` becomes corrupted.

Question 9 below refers to a system comprising two active objects—`actor1` and `actor2`—each hosted by a dedicated thread of control. **Figure 7** depicts an interaction in which these actors attempt (concurrently) to pull an element off a `MonitorQueue` object, which implements its `pull` operation invoking a sequence of operations on another object `q` of type `deque<string>`. As its name suggests, the `MonitorQueue` object is implemented according to the monitor-object pattern.

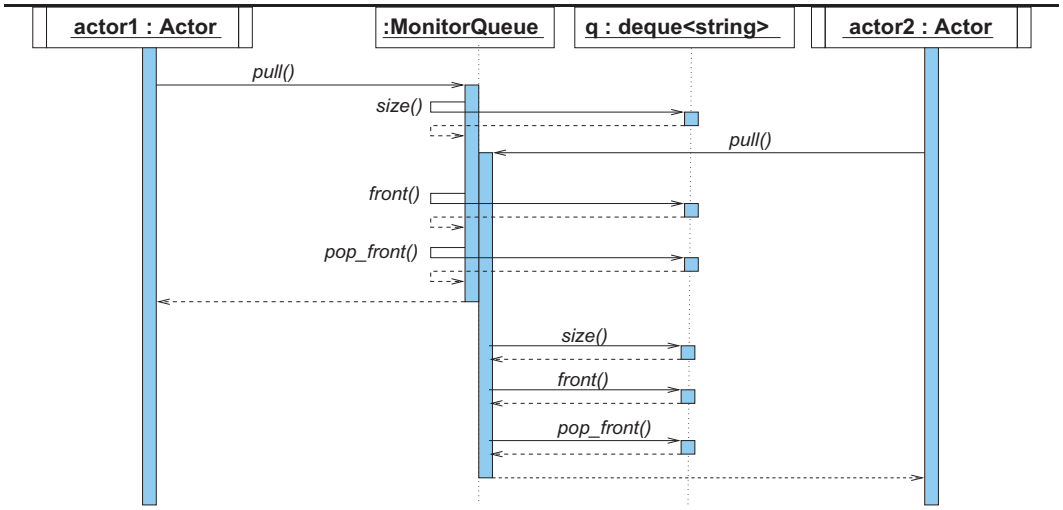


Figure 7: Two actors accessing a queue implemented as a monitor

9. Why does `actor2` delay after invoking the `pull` method and before invoking `size` as depicted in **Figure 7**?

   (a) `actor1` is in the monitor; thus no other thread may execute until `actor1` leaves the monitor.

   (b) `actor2` was delayed because the thread running within `actor1` was switched in to execute just prior to `actor2`'s invocation of `size`. Had this not happened, `actor2` could proceed to execute `size` before `actor1` completes its invocation of the `pull` method.

   (c) `q` is empty and `actor2` must wait until a new object is placed onto `q`.

   (d) The diagram is in error: `actor2` could not send the `pull` message while `actor1` is running in `MonitorQueue`.

   (e) Because `actor1` holds the lock on `MonitorQueue`, `actor2` must wait for `actor1` to release the lock.

Question 10 below refers to a system comprising two active objects—a `LineProducer` and a `LineConsumer`—each hosted by a dedicated thread of control. These actors synchronize with one another via a bounded buffer object `BoundedBuffer`, which is a monitor object that implements its `putLine` and `getLine` operations by invoking a sequence of operations on another object `buf` of type `deque<string>`. Moreover, `BoundedBuffer` was designed to hold at most 3 strings, and it uses condition variables `fullCond` and `emptyCond` to synchronize producers who should block if they execute `putLine` when `buf` is full and consumers who should block if they execute `getLine` when `buf` is empty.

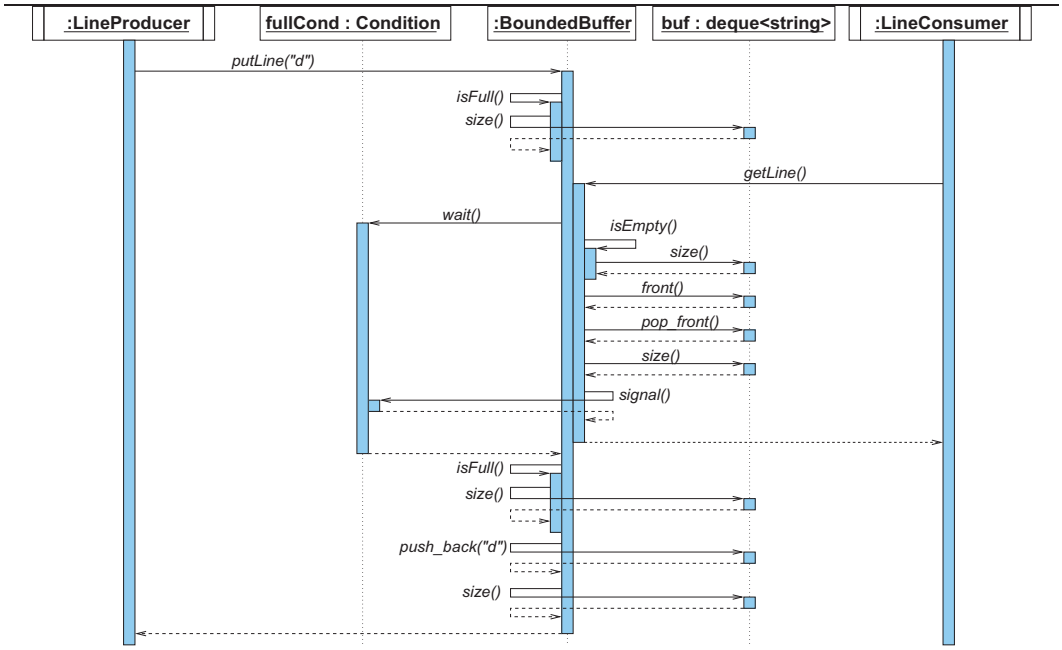


Figure 8: LineProducer and LineConsumer sharing a bounded buffer.

10. Assume that `buf` contains the strings “a”, “b”, and “c” in that order at the beginning of this interaction. Which of the following is true at the end?

    (a) `LineConsumer` gets a copy of string “a”; `buf` contains the strings “b”, “c”, and ”d” in that order.

    (b) `LineConsumer` gets a copy of string “a”; `buf` becomes corrupted.

    (c) The diagram is in error: One thread cannot invoke `signal` on a condition variable while that object is actively servicing `wait` on behalf of another thread.

    (d) The diagram is in error: The `getLine` activation could not have executed `isEmpty`, `front`, `pop_front`, or `size` as depicted because `BoundedBuffer`is a monitor and, when the `getLine` activation begins, the thread hosting the `putLine` activation holds the lock on the monitor.

    (e) The diagram is in error: Because `buf` is full, the `putLine` activation should block immediately, unable to even enter the monitor.

Question 11 below refers to a system comprising two active objects—a `LineProducer` and a `LineConsumer`—each hosted by a dedicated thread of control. These actors synchronize with one another via a bounded buffer object `BoundedBuffer`, which is a monitor object that implements its `putLine` and `getLine` operations by invoking a sequence of operations on another object `buf` of type `deque<string>`. Moreover, `BoundedBuffer` was designed to hold at most 3 strings, and it uses condition variables `fullCond` and `emptyCond` to synchronize producers who should block if they execute `putLine` when `buf` is full and consumers who should block if they execute `getLine` when `buf` is empty.

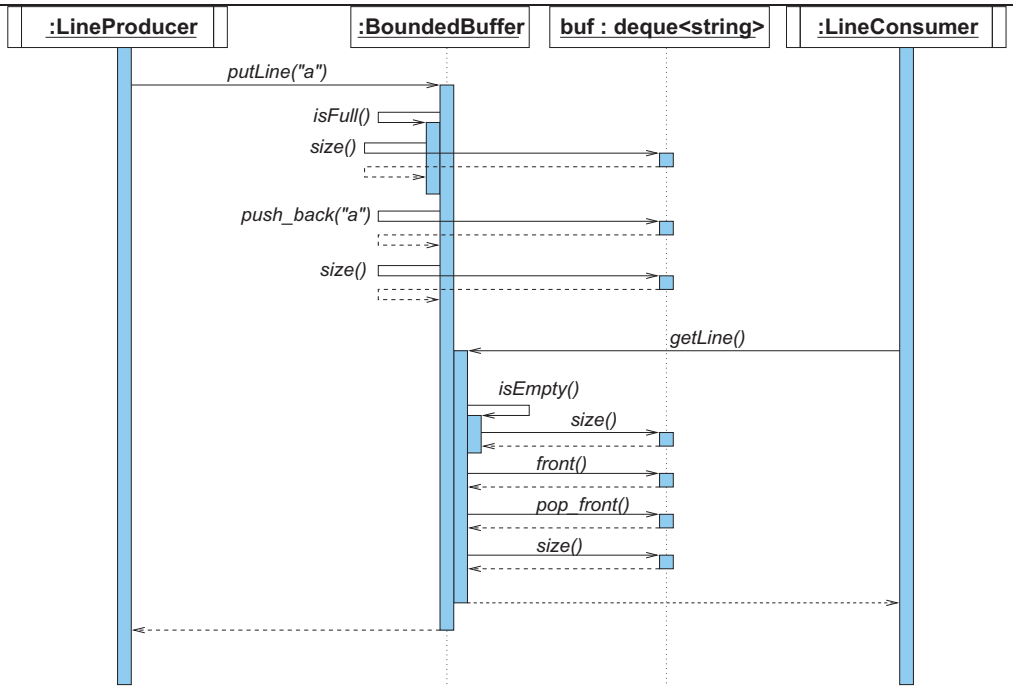


Figure 9: Buffer is initially empty; assumes SharedBuffer is a monitor

11. Assume that `buf` is empty at the beginning of the interaction depicted in **Figure 9**. Which of the following is true?

    (a) Because `buf` is initially empty, `LineConsumer`'s `getLine` message could not possibly have arrived any earlier than is depicted in the diagram.

    (b) The invocation of `push_back` made by the activation of `putLine` implicitly signals `LineConsumer` that the buffer is now receptive to `getLine` messages

    (c) The diagram is in error: Because the activation of `putLine` changed the state of the buffer from empty to non-empty, the `emptyCond` condition variable should have been signaled following `putLine`'s call to `size`.

    (d) The diagram is in error: `LineProducer` should have blocked because `buf` is empty and, when `putLine` was invoked, there was no `LineConsumer` blocked waiting to receive the string that `LineProducer` was attempting to share.

    (e) None of the above.

12. Briefly describe what can go wrong when multiple actors access shared data without synchronization.

13. Which of the following does a monitor guarantee?

    (a) Only one thread at a time may execute within the monitor.

    (b) Once a thread T enters the monitor, no other thread may execute within any object until T exits the monitor.

    (c) Multiple threads may enter a monitor concurrently provided that none of these threads modifies any of the data associated with the monitor object.

    (d) All of the above.

    (e) None of the above.

C.4 saUML vs. standard UML experiment II: post-test

C.4.1 Experiment II: post-test questions

CSE 335 Extra Credit Option 1

Part II: Post-test in conjunction with user study

Name: _____

Questions 1 through 11 reference ten different scenarios of interaction among concurrent actors that access a shared Database object. These actors in this example perform transactions that invoke a a series of methods on the Database object, and we distinguish two different types of actors—*readers* and *writers*. To illustrate, suppose that the Database is storing bank-account information and provides methods for depositing funds, withdrawing funds, and checking the balance of accounts given their account numbers. A typical reader might be interested in computing the total balance of a list of accounts and thus might execute the sequence of operations:

```
unsigned sum=0;
for(unsigned i=0; i < 10; i++) {
   sum += db->getBalance(accounts[i]);
}
```

On the other hand, a writer will perform a transaction that modifies the contents of the Database. For example, a writer client might perform a transaction that transfers $50.00 between accounts $acct_1$ and $acct_2$ by executing the sequence of operations:

```
db->withdraw( acct1, 50);
db->deposit( acct2, 50);
```

Assuming the Database is implemented as a monitor, it should be safe for multiple reader transactions to execute concurrently because reader clients do not modify the contents of the Database object. However, a writer transaction should never execute concurrently with any reader or any other writer transaction. Class database supports this *readers-writer* style of synchronization by providing four methods—`startRead()`, `stopRead()`, `startWrite()`, and `stopWrite()`—which reader and writer threads use to signal the start and finish of one of these transactions.

The first attachment to this test contains the C++ code for classes `Reader`, `Writer`, and `Database`. Notice that the "account management" operations for class `Database` have been elided here for brevity. Class `Database`:

- is implemented according to the monitor-object pattern, using the private variable `lock_` as the monitor lock;
- defines two counting variables, `nReaders_` and `nWriters_`, which record the number of concurrently executing reader and writer transactions respectively; and

- defines two condition variables, `okToRead_` and `okToWrite_`, which are used to synchronize reader and writer threads as they begin and end their transactions.

Please take a moment to familiarize yourself with this code.

Scenario 1 (an interaction involving one reader and one writer): Assume the reader thread is running within the invocation of `startRead()`, and the writer thread is in the ready state. A context switch occurs just after the reader thread increments `nReaders_` by one. The reader thread transitions to ready and the writer thread transitions to running. The writer thread invokes `startWrite()`.

1. Shortly thereafter:
   A. The writer thread obtains the monitor lock.
   B. The reader thread suspends but does not release the lock.
   C. The writer thread suspends.
   D. Both the reader thread and the writer suspend and deadlock occurs.
   E. None of the above

Scenario 2 (an interaction involving one reader and one writer): Assume the reader thread is in the running state and the writer thread is in the suspended state (we assume the writer thread is suspended on `okToWrite_.wait()`. The reader thread invokes endRead() and enters the monitor. It sets the `nReaders_` to 0 and issues a okToWrite`.signal()`.

2. As a result:
   A. The writer thread remains suspended.
   B. The writer thread transitions to ready and acquires the monitor lock.
   C. The writer thread transitions to ready but does not yet acquire the monitor lock.
   D. The writer thread transitions to running and acquires the monitor lock.
   E. The writer thread transitions to running and does not acquire the monitor lock.
3. Upon completing the invocation okToWrite`.signal()`, the reader thread:
   A. Must change state to ready and release the monitor lock
   B. Must change state to ready and retain the monitor lock
   C. May remain running and must retain the monitor lock
   D. Must remain running and may release the monitor lock
   E. Must suspend and release the monitor lock

Scenario 3 (an interaction involving one reader and one writer): Assume that, after the reader thread has returned from its invocation of `startRead()`, a context switch occurs, and the writer thread invokes `startWrite()`.

4. Shortly after the writer thread issues the `startWrite()` message:
   A. The writer thread remains running until the invocation completes.
   B. The writer thread suspends on the monitor lock.
   C. The writer thread obtains the monitor lock but then suspends shortly thereafter.
   D. The reader thread suspends.
   E. Deadlock occurs.

Scenario 4 (an interaction involving one reader and one writer): Assume the writer thread is running within the invocation of `startWrite()` and the reader thread is in the ready state. A context switch occurs just after the writer thread increments `nWriters_` by one. The writer thread transitions to ready and the reader thread transitions to running. The reader thread invokes `startRead()`, but suspends afterwards.

5. Why does the reader thread suspend?
   A. The reader thread suspends on `okToWrite_.wait()`, since `nWriters_` is non-zero at the time.
   B. Deadlock occurs. Since the writer thread is in the monitor, the reader thread can't possibly enter the monitor.
   C. The reader thread suspends on the monitor lock.
   D. The reader thread suspends on `okToRead_.wait()`, since `nWriters_` is non-zero at the time.
   E. The reader thread suspends due to the occurrence of a context switch.

Scenario 5 (an interaction involving two readers): Assume the reader thread (r1) has completed the invocation of `startRead()` and the other reader thread (r2) is in the ready state. A context switch occurs. r1 transitions to ready and r2 transitions to running. r2 issues a `startRead()` message.

6. What will happen as a result of this invocation of `startRead()`?
   A. r2 enters the monitor but suspends on `okToRead_.wait()`, since `nReaders_` is non-zero at the time.
   B. r2 enters the monitor but suspends on `okToRead_.wait()`, since `nWriters_` is non-zero at the time.
   C. r2 suspends on the monitor lock.
   D. r2 enters the monitor and increases `nReaders_` to two.
   E. r2 suspends due to the occurrence of a context switch.

Scenario 6 (an interaction involving one reader and one writer): Assume the reader thread is running within the invocation of `startRead()` and that the writer thread, having issued a call to `startWrite()` is suspended on the monitor lock.

7. When the reader thread returns from `startRead()`, thus releasing the monitor lock:
   A. The reader thread must transition to ready; the writer thread must transition to running.
   B. The reader thread must transition to suspended; the writer thread must transition to running.
   C. The reader thread may remain running; the writer thread must remain suspended.

D. The reader thread may remain running; the writer thread must transition to ready.

E. Deadlock occurs.

Scenario 7 (an interaction involving one reader and one writer): Assume the writer thread has completed its invocation of `startWrite()` and the reader thread is in the ready state. A context switch occurs. The writer thread transitions to ready and the reader thread transitions to running. The reader thread issues a `startRead()`.

8. Shortly thereafter:

   A. The reader thread completes the invocation of `startRead()` without suspending.

   B. The writer suspends because the reader has entered the monitor.

   C. The reader thread suspends on the monitor lock.

   D. The reader thread suspends on `okToRead_.wait()`.

   E. Both 8B and 8D are true; thus deadlock occurs.

Scenario 8 (an interaction involving one reader and one writer): Assume the reader thread is running within the invocation of endRead() and the writer thread, having issued an invocation of `startWrite()` is now suspended on the monitor lock. The reader thread invokes okToWrite`.signal()`.

9. As a result of the signal:
   A. The reader thread releases the monitor lock; the writer thread transitions to running.
   B. The reader thread releases the monitor lock; the writer thread transition to ready.
   C. The reader thread retains the monitor lock; the writer thread remains suspended.
   D. The reader thread retains the monitor lock; the writer thread transitions to ready.
   E. Deadlock occurs.

Scenario 9 (an interaction involving two reader threads): Assume the reader thread (r1) is running within the invocation of `startRead()` and the other reader thread (r2) is in the ready state. A context switch occurs just after r1 increments `nReaders_` by one. r1 transitions to ready and r2 transitions to running. r2 thread invokes `startRead()`.

10. Shortly thereafter:
    A. r2 enters the monitor but suspends because `nReaders_` is non-zero when r2 enters the monitor.
    B. r2 enters the monitor and increases `nReaders_` to two.
    C. r2 suspends on the monitor lock.
    D. r2 completes its invocation of `startRead()` without suspending.
    E. Deadlock occurs.

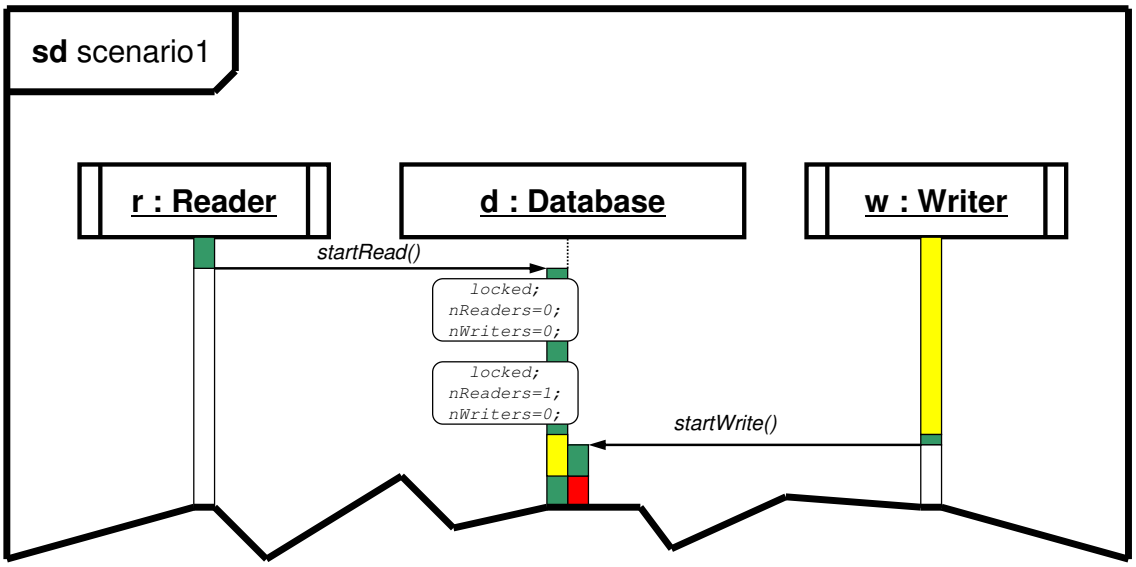
Scenario 10 (an interaction involving two writer threads): Assume the writer thread (w1) has completed its invocation of `startWrite()` and the other writer thread (w2) is in the ready state. A context switch occurs. w1 transitions to ready and w2 transitions to running. w2 issues a `startWrite()`.

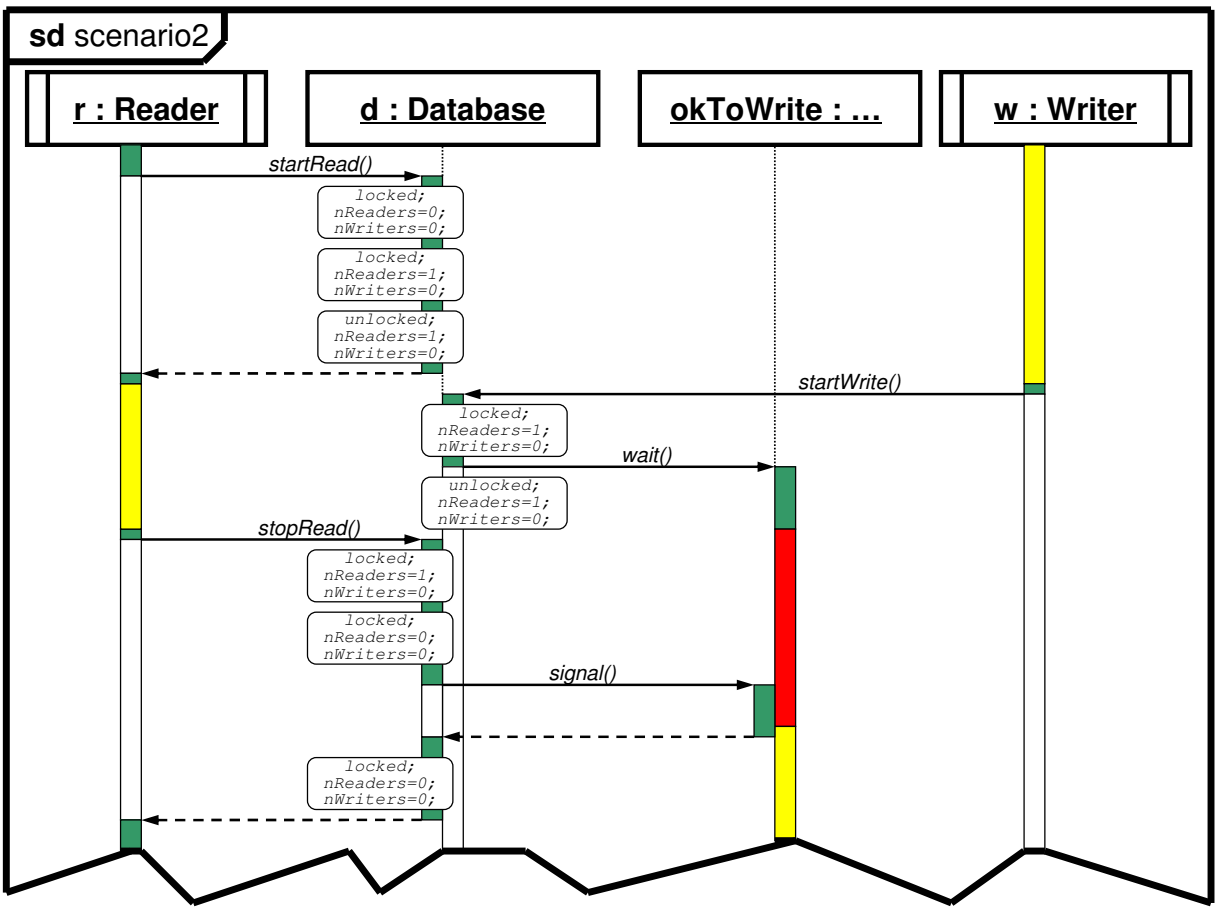
11. As a result of this invocation of `startWrite()`:
    A. w2 enters the monitor and suspends on `okToWrite_.wait()`, retaining the monitor lock.
    B. w2 enters the monitor and suspends on `okToWrite_.wait()`, releasing the monitor lock.
    C. w2 suspends on the monitor lock.
    D. w2 enters the monitor and increases `nWriters_` to two.
    E. Deadlock occurs.

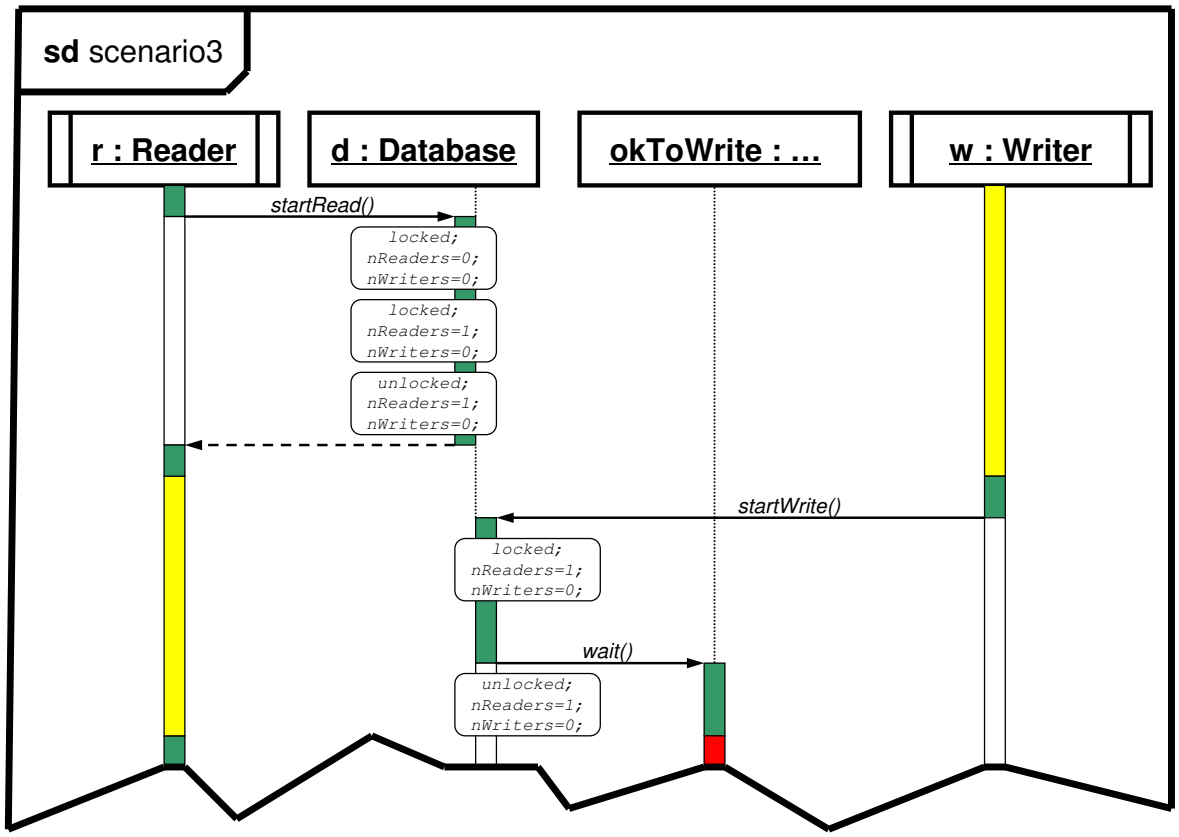
The following questions are not related to any scenario.

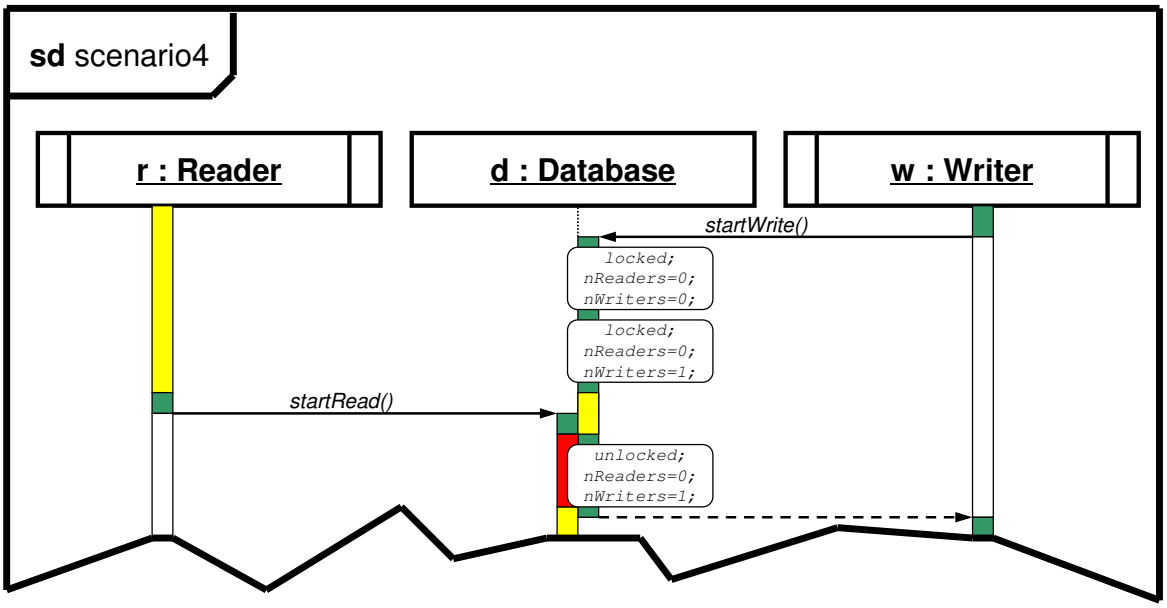
12. What feature(s) of the monitor implementation of class `Database` prevent race conditions in updating counting variables (`nReaders_` and `nWriters_`)?
    A. calls to wait on the condition variables `okToRead_` and `okToWrite_`.
    B. calls to signal or broadcast on the condition variables `okToRead_` and `okToWrite_`.
    C. the fact that calls to wait implicitly release the lock before the calling thread suspends.
    D. the need for any thread to acquire the monitor lock before entering the monitor.
13. What feature(s) of the monitor implementation prevent a writer from entering the Database while reader(s) are present?
    A. calls to wait on the condition variables `okToRead_` and `okToWrite_`.
    B. calls to signal or broadcast on the condition variables `okToRead_` and `okToWrite_`.
    C. the fact that calls to wait implicitly release the lock before the calling thread suspends.
    D. the need for any thread to acquire the monitor lock before entering the monitor.
14. Why does the wait method release the lock and then acquire it again?
    A. To “wake-up” a reader thread that was previously blocked on the wait.
    B. To “wake-up” a writer thread that was previously blocked on the wait.
    C. To promote efficiency.
    D. To prevent deadlock.

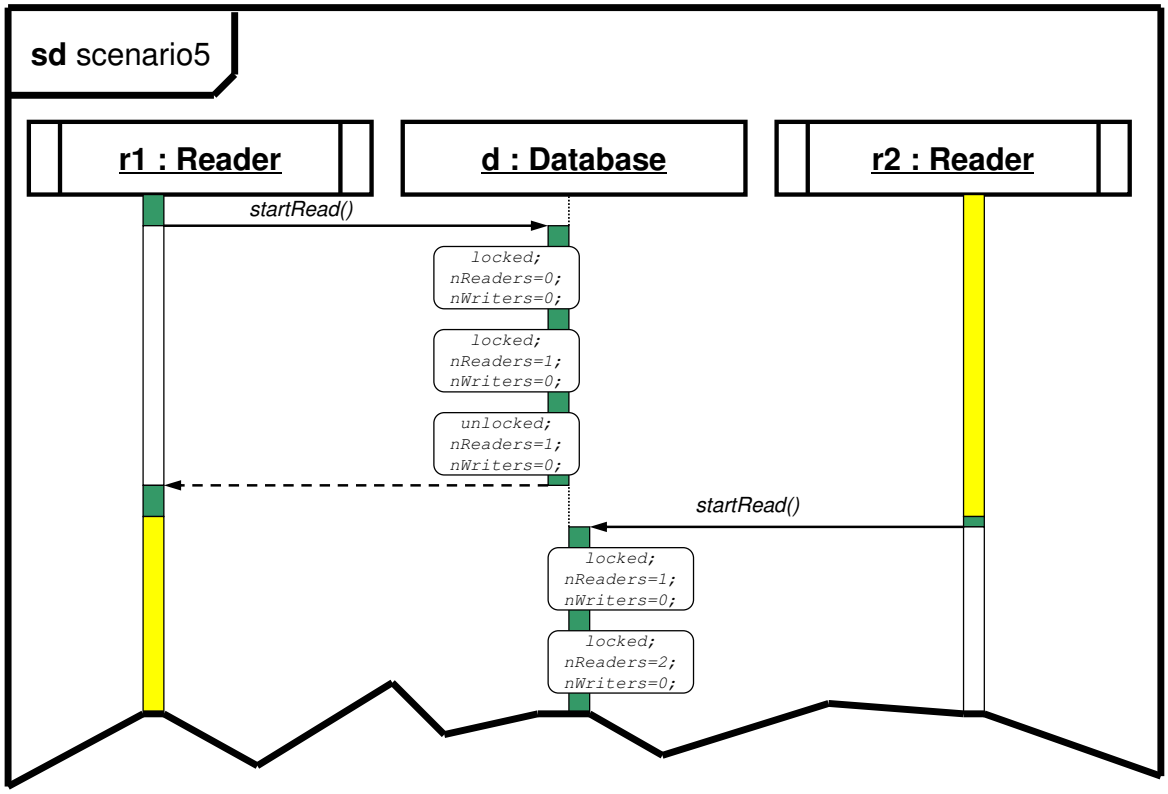
C.4.2 Experiment II: post-test saUML sequence diagrams

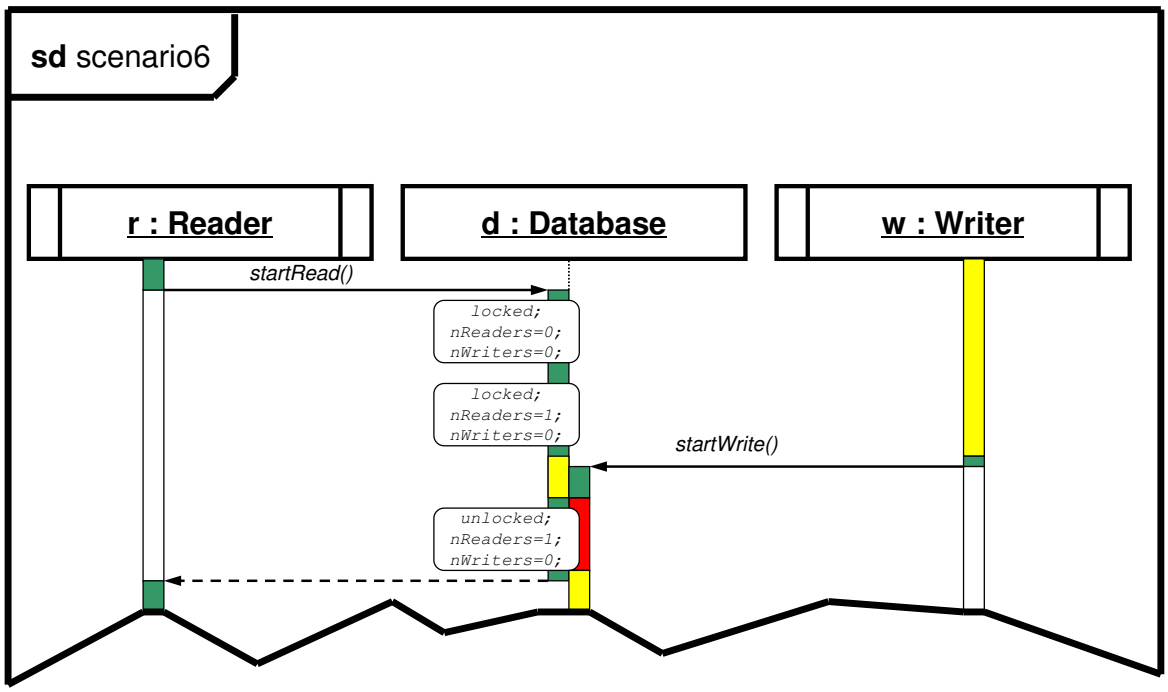

sd scenario1
r : Reader
d : Database
w : Writer
startRead()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=1;
nWriters=0;
startWrite()

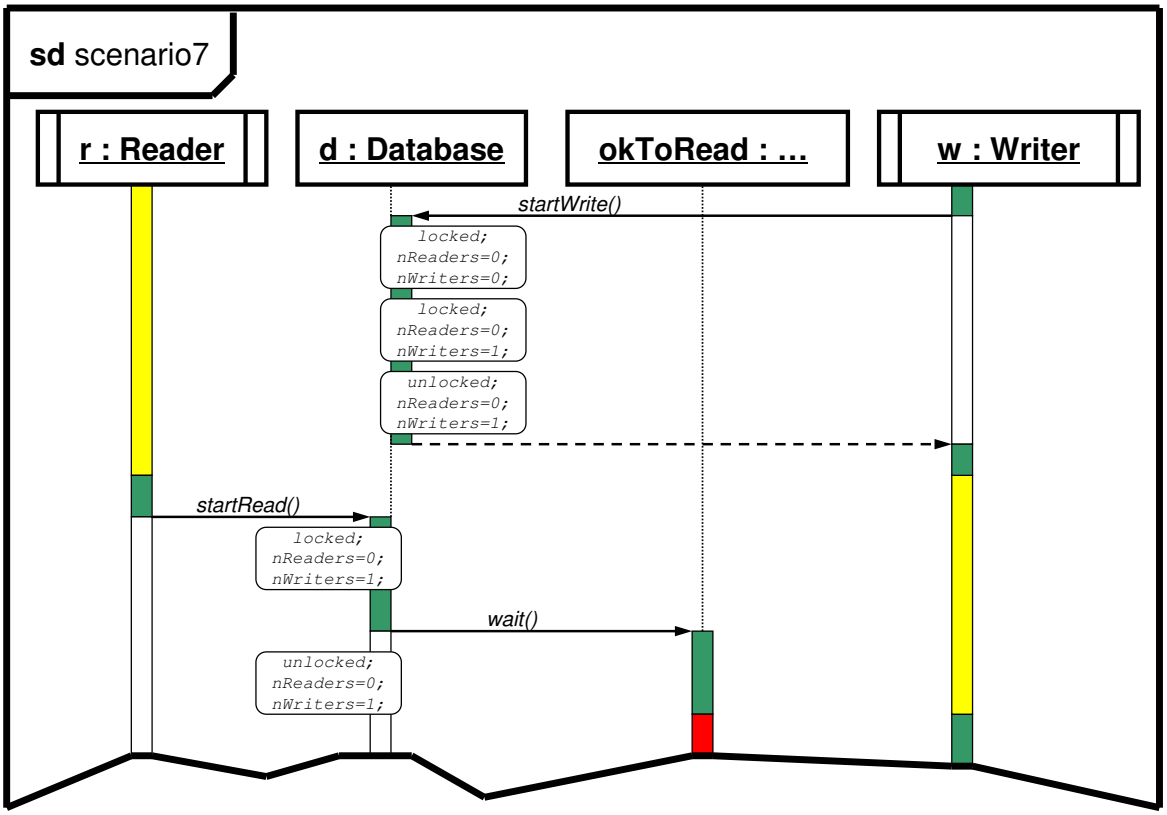

sd scenario2
r : Reader
d : Database
okToWrite : ...
w : Writer
startRead()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=1;
nWriters=0;
unlocked;
nReaders=1;
nWriters=0;
startWrite()
locked;
nReaders=1;
nWriters=0;
wait()
unlocked;
nReaders=1;
nWriters=0;
stopRead()
locked;
nReaders=1;
nWriters=0;
locked;
nReaders=0;
nWriters=0;
signal()
locked;
nReaders=0;
nWriters=0;

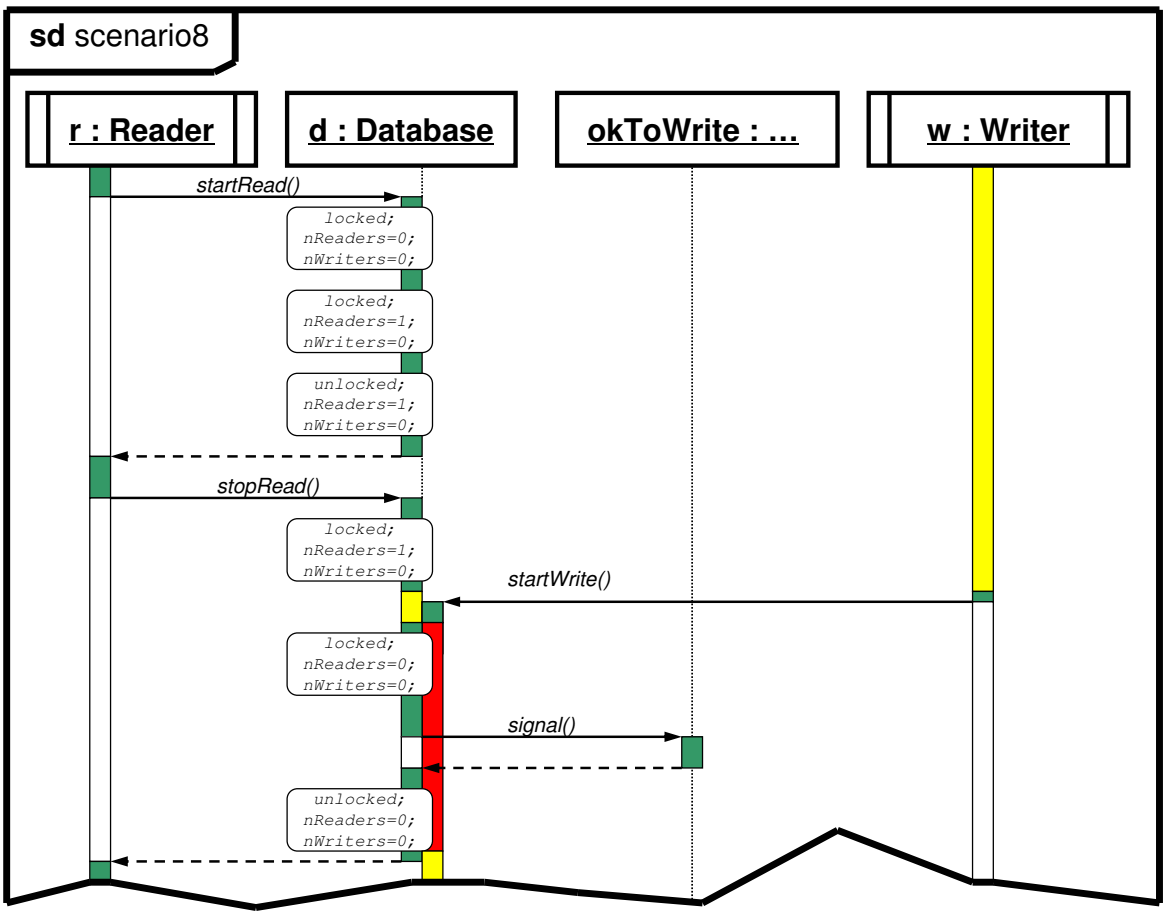

sd scenario3
r : Reader
d : Database
okToWrite : ...
w : Writer
startRead()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=1;
nWriters=0;
unlocked;
nReaders=1;
nWriters=0;
startWrite()
locked;
nReaders=1;
nWriters=0;
wait()
unlocked;
nReaders=1;
nWriters=0;

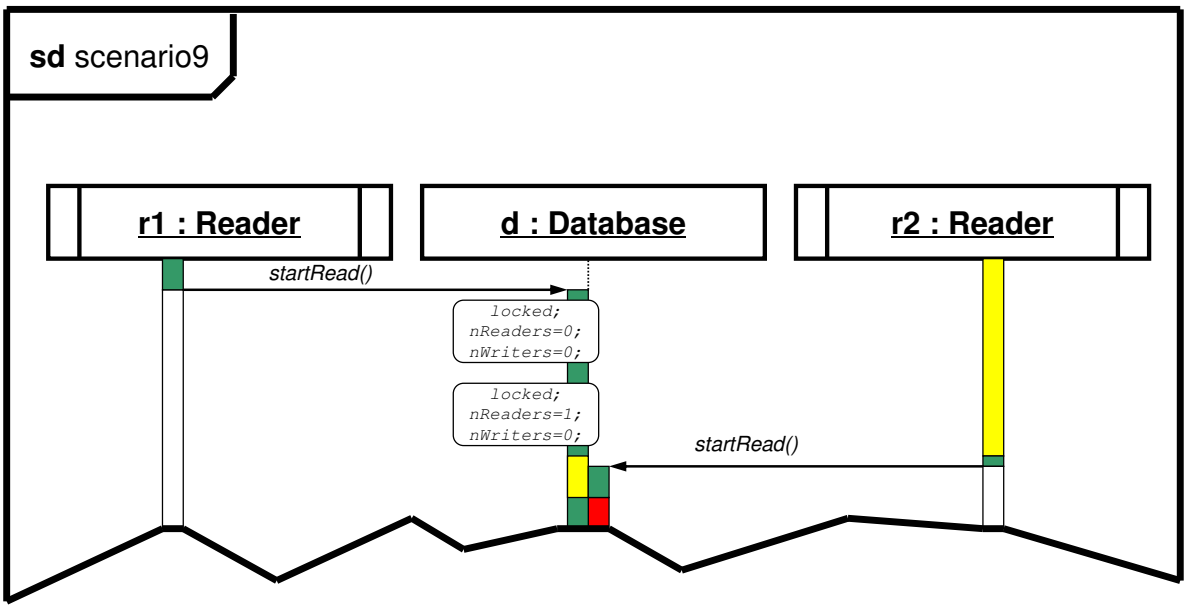

sd scenario4
r : Reader
d : Database
w : Writer
startWrite()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=0;
nWriters=1;
startRead()
unlocked;
nReaders=0;
nWriters=1;

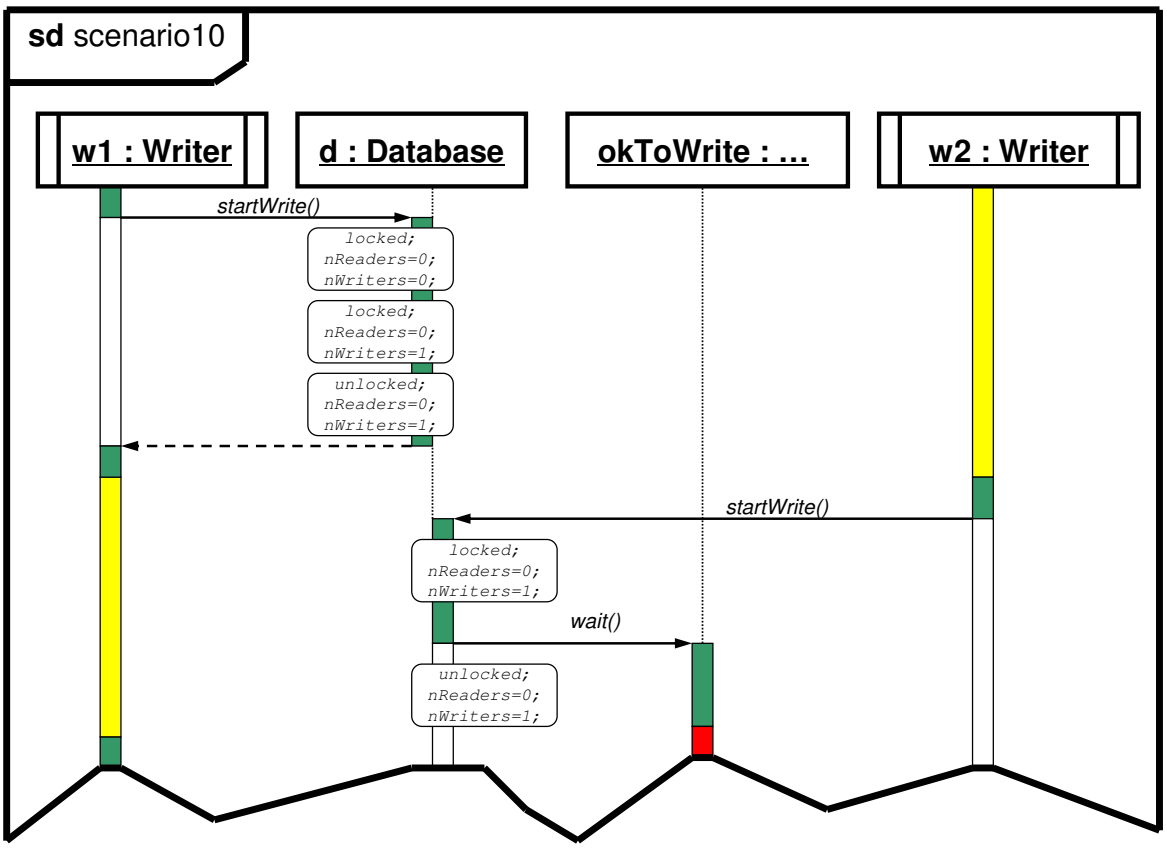

sd scenario5
r1 : Reader
d : Database
r2 : Reader
startRead()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=1;
nWriters=0;
unlocked;
nReaders=1;
nWriters=0;
startRead()
locked;
nReaders=1;
nWriters=0;
locked;
nReaders=2;
nWriters=0;


sd scenario6
r : Reader
d : Database
w : Writer
startRead()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=1;
nWriters=0;
startWrite()
unlocked;
nReaders=1;
nWriters=0;

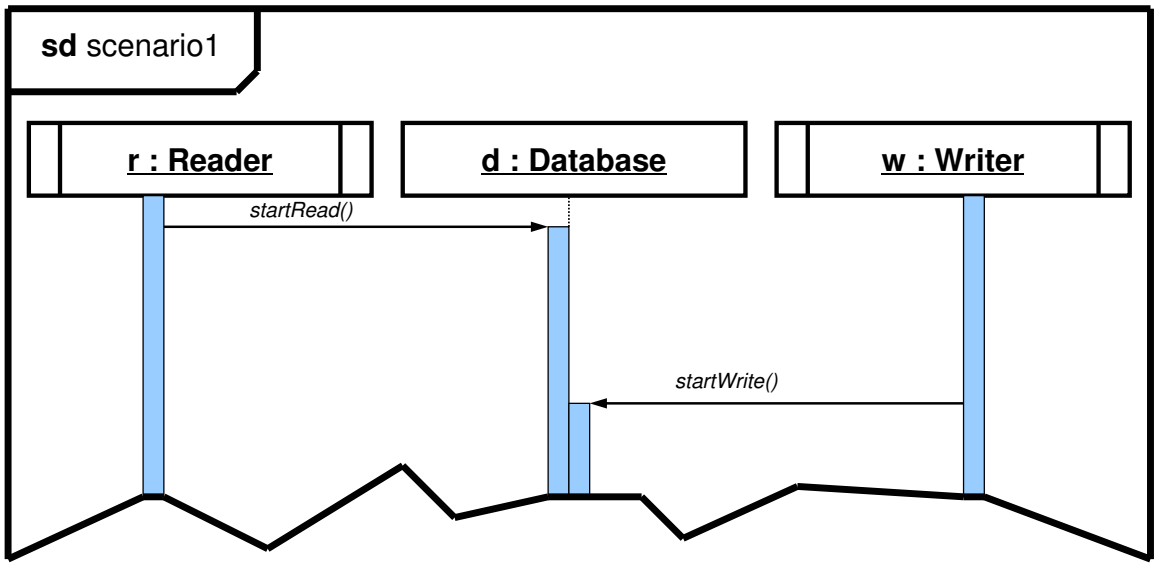

sd scenario7
r : Reader
d : Database
okToRead : ...
w : Writer
startWrite()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=0;
nWriters=1;
unlocked;
nReaders=0;
nWriters=1;
startRead()
locked;
nReaders=0;
nWriters=1;
wait()
unlocked;
nReaders=0;
nWriters=1;

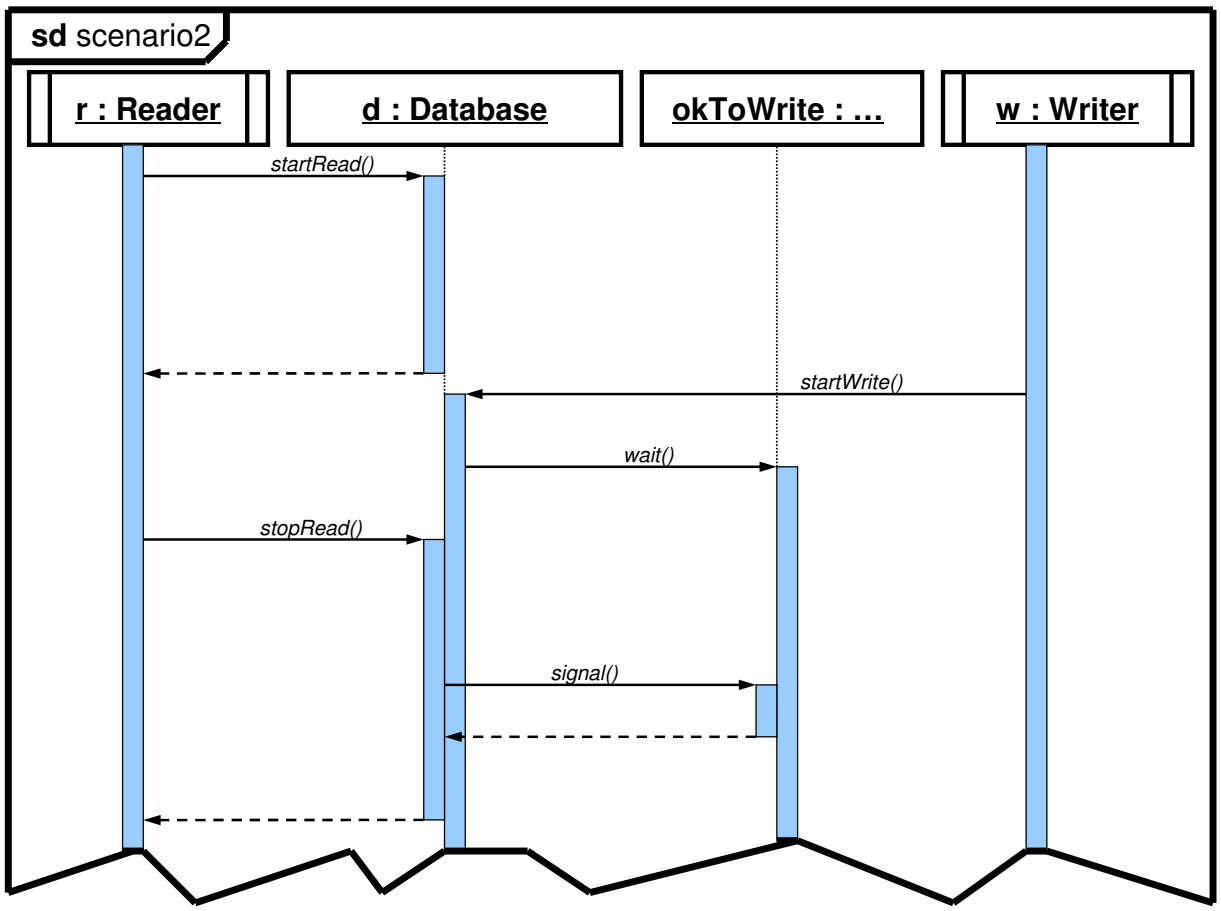

sd scenario8
r : Reader
d : Database
okToWrite : ...
w : Writer
startRead()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=1;
nWriters=0;
unlocked;
nReaders=1;
nWriters=0;
stopRead()
locked;
nReaders=1;
nWriters=0;
startWrite()
locked;
nReaders=0;
nWriters=0;
signal()
unlocked;
nReaders=0;
nWriters=0;

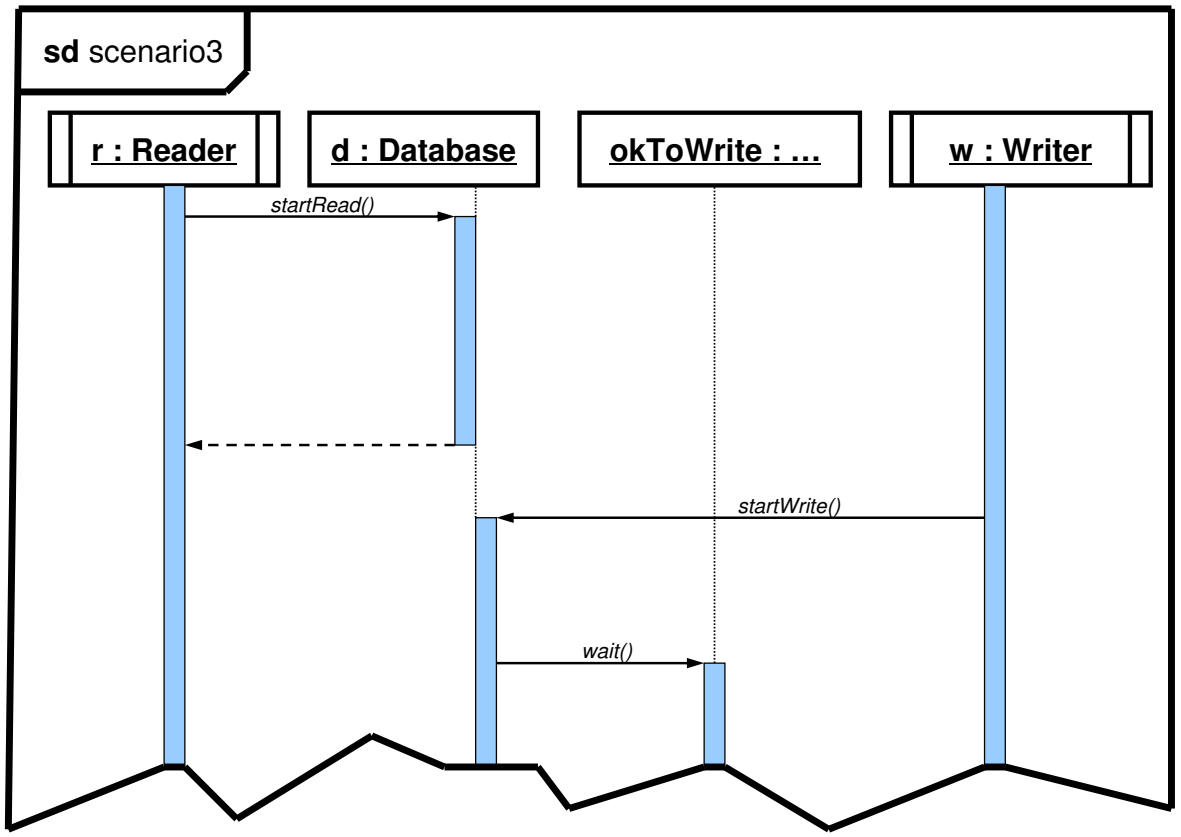

sd scenario9
r1 : Reader
d : Database
r2 : Reader
startRead()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=1;
nWriters=0;
startRead()

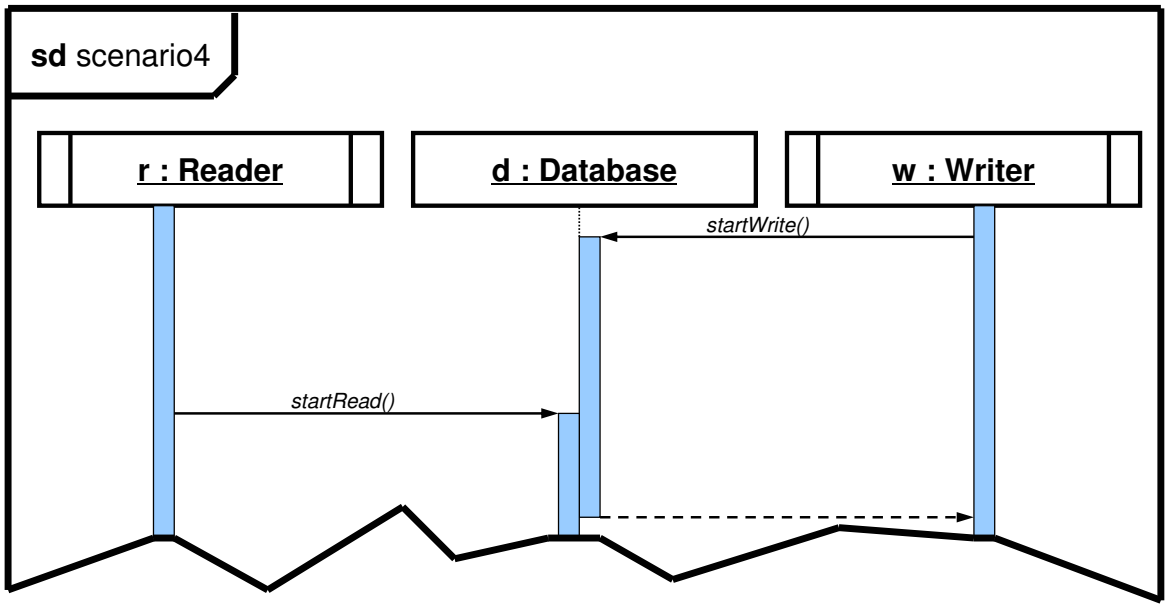

sd scenario10
w1 : Writer
d : Database
okToWrite : ...
w2 : Writer
startWrite()
locked;
nReaders=0;
nWriters=0;
locked;
nReaders=0;
nWriters=1;
unlocked;
nReaders=0;
nWriters=1;
startWrite()
locked;
nReaders=0;
nWriters=1;
wait()
unlocked;
nReaders=0;
nWriters=1;

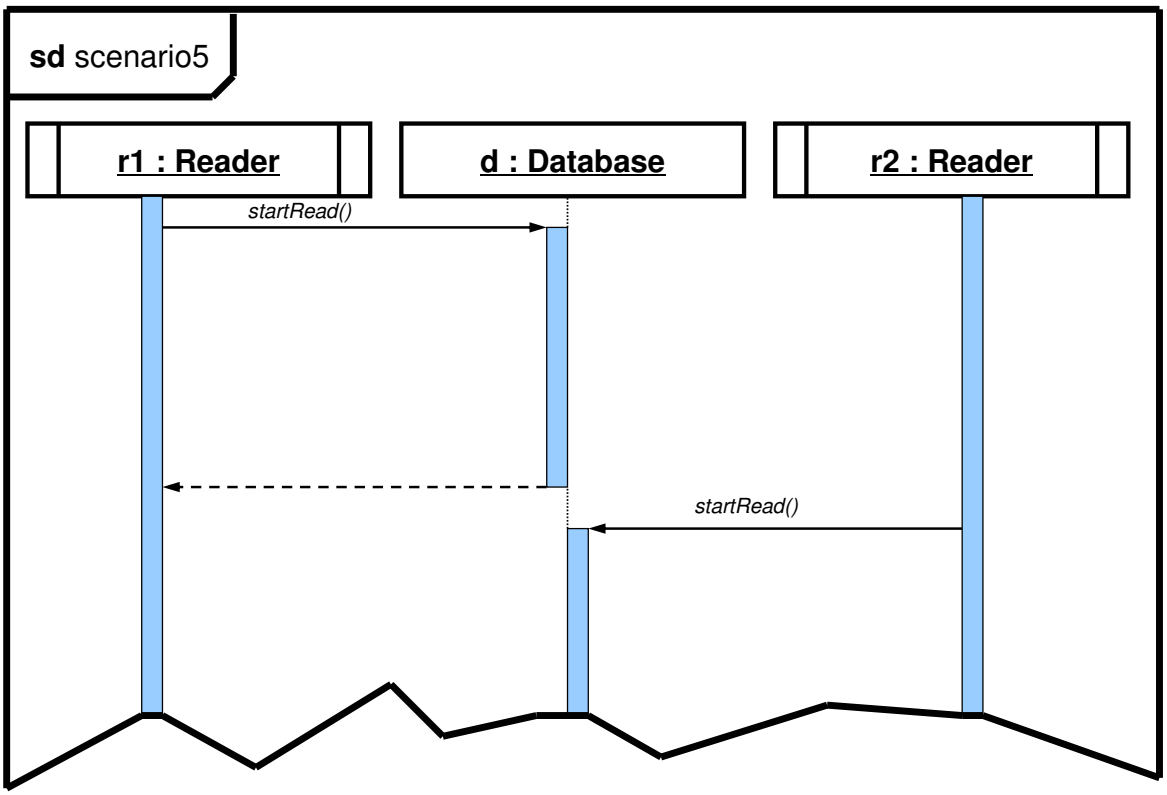
C.4.3 Experiment II: post-test UML sequence diagrams

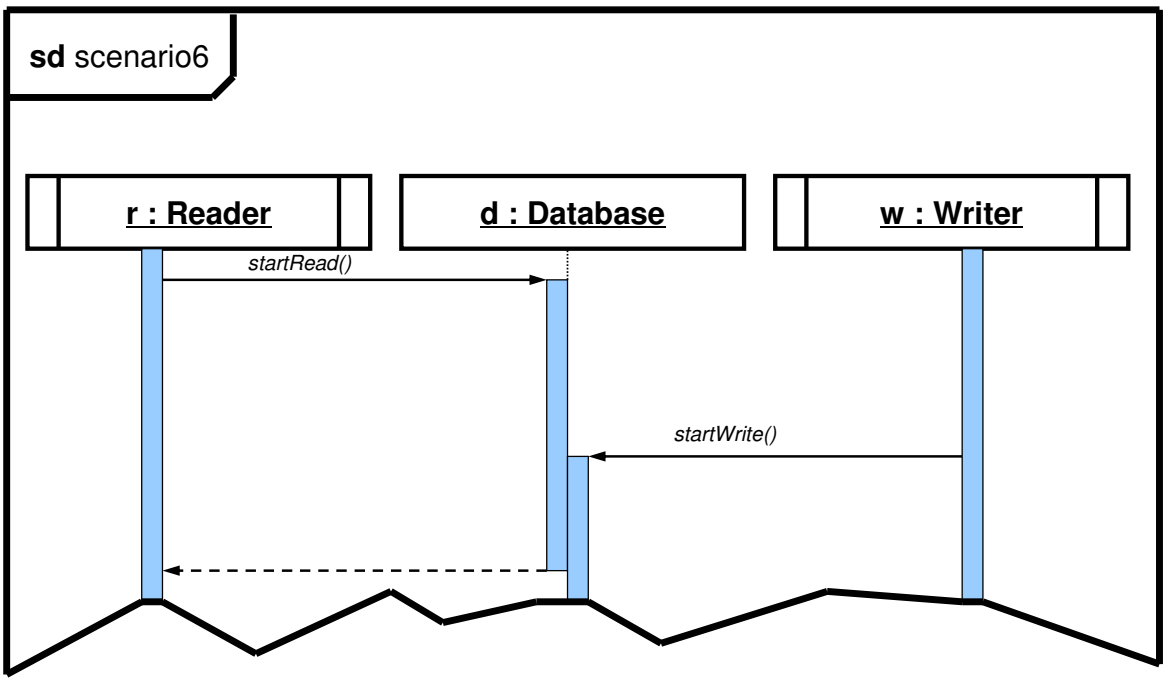

sd scenario1
r : Reader
d : Database
w : Writer
startRead()
startWrite()

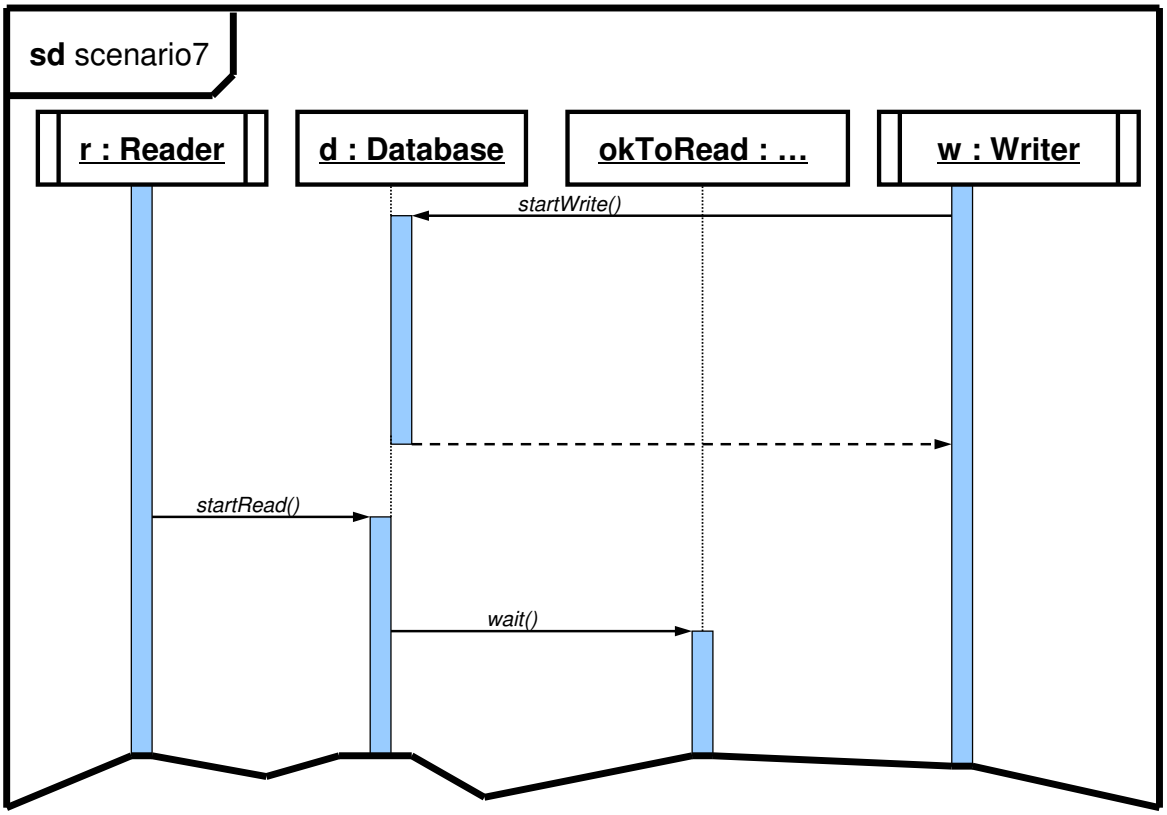

sd scenario2
r : Reader
d : Database
okToWrite : ...
w : Writer
startRead()
startWrite()
wait()
stopRead()
signal()

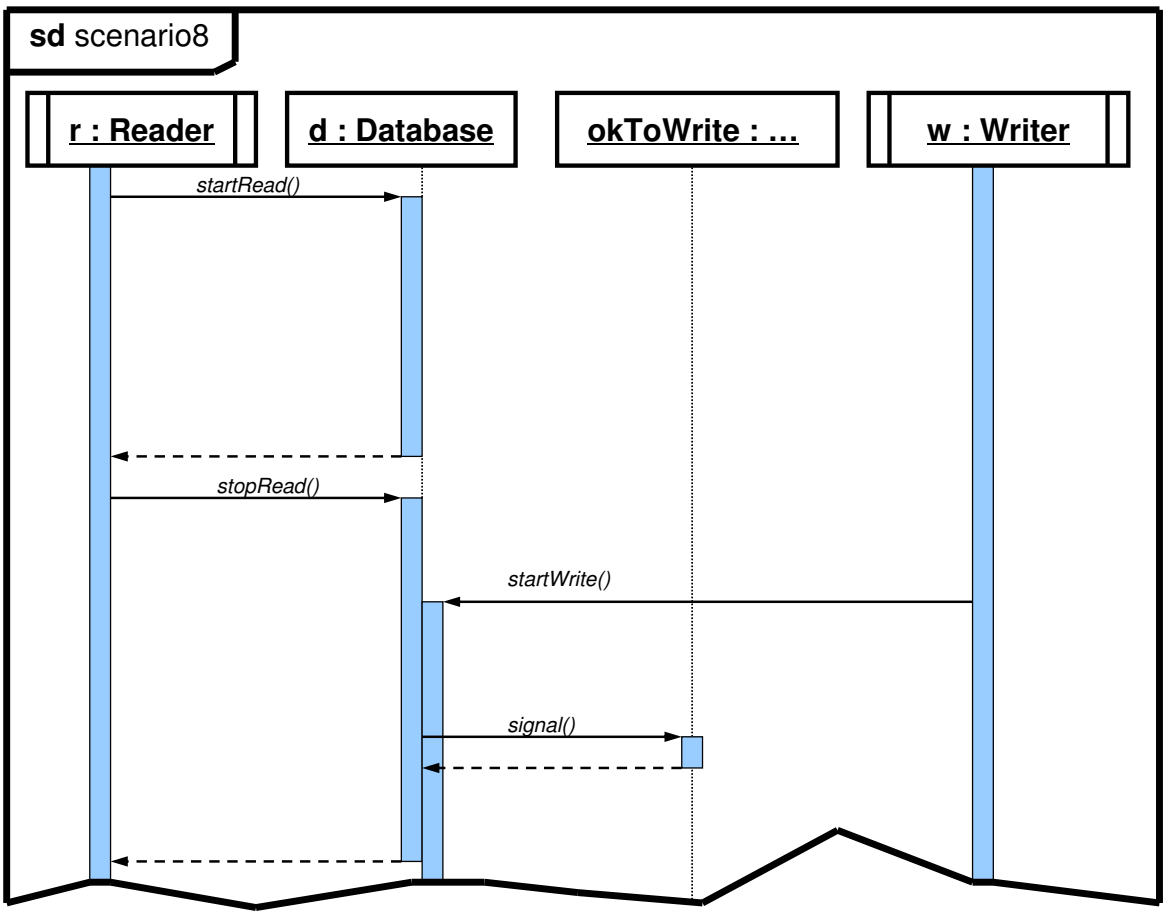

sd scenario3
r : Reader
d : Database
okToWrite : ...
w : Writer
startRead()
startWrite()
wait()

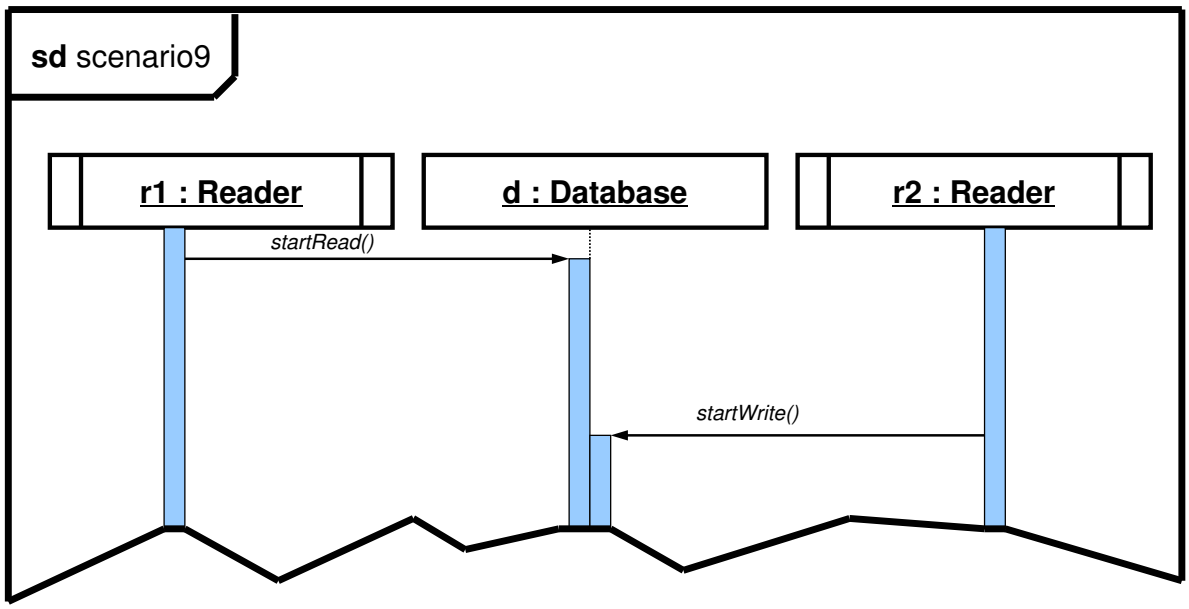

sd scenario4
r : Reader
d : Database
w : Writer
startWrite()
startRead()

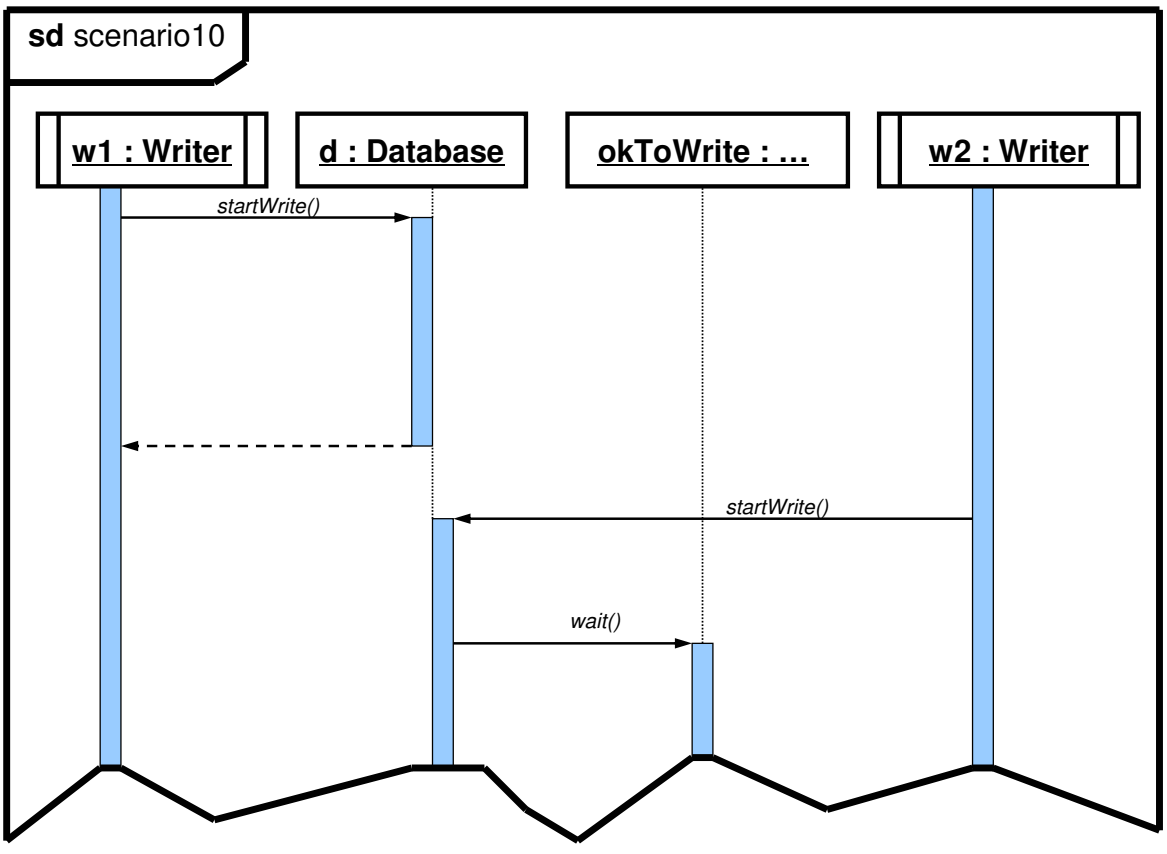

sd scenario5
r1 : Reader
d : Database
r2 : Reader
startRead()
startRead()


sd scenario6
r : Reader
d : Database
w : Writer
startRead()
startWrite()


sd scenario7
r : Reader
d : Database
okToRead : ...
w : Writer
startWrite()
startRead()
wait()


sd scenario8
r : Reader
d : Database
okToWrite : ...
w : Writer
startRead()
stopRead()
startWrite()
signal()


sd scenario9
r1 : Reader
d : Database
r2 : Reader
startRead()
startWrite()


sd scenario10
w1 : Writer
d : Database
okToWrite : ...
w2 : Writer
startWrite()
startWrite()
wait()

Appendix D

Materials used in the state machine modeling vs. non-modeling study

D.1 State machine modeling vs. non-modeling : Pre-test questions

Evaluating UML 2.0 state diagrams: Pre-test

November 26, 2007

1. A *monitor* is:
   (a) an operation that must be performed entirely or not performed at all.
   (b) a sequential stream of execution.
   (c) a class-like programming language construct within which only one thread may execute concurrently.
   (d) a section of code that must be executed atomically.
   (e) a section of code in which the result of computation depends on the ordering of thread execution.
2. A race condition is:
   (a) the situation in which multiple threads or processes read and write a shared data item and the final result depends on the order of execution.
   (b) the situation where no progress can be made because each thread or process is waiting for an event that only another process or thread in the set can cause.
   (c) the condition in which only one thread at a time may access a shared resource.
   (d) an operation that must be performed entirely or not performed at all.
   (e) a section of code that must be executed atomically.
3. A critical section is:
   (a) a section of code that may be accessed by only one thread at a time.
   (b) a sequential stream of execution.
   (c) a variable used to synchronize threads.
   (d) an operation that suspends/blocks execution of the calling process if a certain condition is true.
   (e) an operation that resumes the execution of some suspended thread or process by placing it in the processor ready queue.

4. Which of the following is true of the typical use of condition variables in a monitor?

   (a) Invoking wait() on a condition variable suspends the invoking thread.

   (b) Invoking signal() on a condition variable resumes a thread waiting on that condition variable.

   (c) If there is no waiting thread, a signal can be stored/delayed to resume the next thread that waits on that condition variable.

   (d) a and b only

   (e) a, b, and c

5. Which of the following description(s) is true?

   (a) Acquiring a mutex lock takes place before anything else in a monitor method.

   (b) Releasing a mutex lock takes place after everything else in a monitor method.

   (c) The monitor mutex lock must be both acquired and released in a monitor method to ensure correct behavior.

   (d) a and b only

   (e) a, b, and c

6. What feature of the monitor implementation prevents race conditions in updating counting variables?

   (a) The evaluation of a condition.

   (b) The wait() operation on condition variables.

   (c) The lock on the monitor.

   (d) The signal() operation on condition variables.

   (e) That a thread releases the lock prior to suspending in the wait() operation.

7. In a monitor implementation, the monitor lock serves the following purpose:

   (a) To prevent race conditions among threads which are sharing access to the monitor.

   (b) To prevent deadlock among threads which are sharing access to the monitor.

   (c) To ensure fairness among threads which are sharing access to the monitor.

   (d) All of the above.

   (e) None of the above.

8. In the implementation of a monitor that employs condition synchronization, which of the following features contributes to deadlock avoidance among threads that share the monitor?

   (a) Threads resumed by `signal()` reacquire the monitor lock prior to returning from `wait()`.

   (b) Threads release the monitor lock prior to suspending in the `wait()` operation.

   (c) Threads acquire the monitor lock upon entry into a method of the monitor.

   (d) A thread blocks while trying to acquire the monitor lock when another thread holds the lock.

   (e) None of the above.

9. Which of the following does a monitor guarantee?

   (a) Once a thread T enters the monitor, no other thread may execute any methods within the monitor until T returns from that method.

   (b) Only one thread at a time may execute within the monitor.

   (c) Multiple threads may enter a monitor concurrently provided that none of these threads modifies any of the data associated with the monitor object.

   (d) a and c only

   (e) a and b only

Figure 1: UML 2.0 state model of a Bounded Buffer monitor object

Questions 10 through **??** refer to a UML 2.0 state model of a BoundedBuffer monitor object, as depicted in Figure 1.

10. In the diagram of Figure 1, the label "[q.size $<$ MAX]" is known as a(n):

    (a) action

    (b) activity

    (c) call

    (d) guard

    (e) state label

11. The BoundedBuffer enters the "Idle" state

    (a) upon creation

    (b) upon exiting the Pushing state

    (c) upon exiting the Pulling state

(d) b and c

(e) a, b, and c

12. When a client process calls pull() on an instance of the BoundedBuffer process,

    (a) the BoundedBuffer process stops looping on q.pullFront(), exits the Pulling state, and then replies with the result of q.pullFront().

    (b) If (q.size > 0), the BoundedBuffer process accepts the pull() call and enters the Pulling state, where it loops on q.pullFront() until a response is generated. It then replies to the calling process with that response.

    (c) If (q.size > 0), the BoundedBuffer process accepts the pull() call and enters the Pulling state, where it executes and obtains a reply from q.pullFront(). It then replies to the calling process with that response.

    (d) The BoundedBuffer process accepts the call to pull() and then checks the condition (q.size > 0). If the condition is true, the BoundedBuffer process enters the Pulling state, where it loops on q.pullFront() until a response is generated. It then replies to the calling process with that response.

    (e) The BoundedBuffer process accepts the call to pull() and then checks the condition (q.size > 0). If the condition is true, the BoundedBuffer process enters the Pulling state, where it executes and obtains a reply from q.pullFront(). It then replies to the calling process with that response.

13. Assume that this BoundedBuffer is shared by multiple client processes, that MAX is 10 and the current queue size is 1. What will happen if three client processes invoke push(x) calls on the BoundedBuffer at virtually the same time?

    (a) The BoundedBuffer process will enter the Pushing state with a count of 3.

    (b) One client process will enter the Pushing state and two client processs will remain in the Idle state.

    (c) The BoundedBuffer process will accept the push(x) call of the first client process and enter the Pushing state. The other calls will be queued up until the pushBack(x) call is performed.

    (d) The BoundedBuffer process will accept the push(x) call of the first client process and enter the Pushing state. The other calls will be queued up until the BoundedBuffer has issued a reply and entered the Idle state.

(e) The BoundedBuffer will accept the push(x) call of the first client process and enter the Pushing state. The other calls will be queued up until the BoundedBuffer has accepted a call to pull(), entered the Pulling state, and then issued a reply and entered the Idle state.

14. Assume that this BoundedBuffer is shared by multiple threads, that MAX is 10 and the current queue size is 10. What will happen if a client process invokes a push(x) call on the BoundedBuffer?

    (a) The BoundedBuffer will reject the push(x) call.

    (b) The BoundedBuffer will queue up the push(x) call until (q.size $<$ MAX), thus blocking the client process.

    (c) The BoundedBuffer will queue up the push(x) call until (q.size $<$ MAX). The client process may continue.

    (d) The BoundedBuffer will accept the push(x) call and then evaluate the condition (q.size $<$ MAX). The client process will block.

    (e) The BoundedBuffer will accept the push(x) call and then evaluate the condition (q.size $<$ MAX). The client process will continue.

15. In a rendezvous style communication, which of the following description(s) is true?

    (a) After sending a request to a server process, a client process blocks until the server replies.

    (b) Upon completion of serving a request from a client process, a server process sends a reply to the client process.

    (c) When replying to a request from a client process, a server process blocks until the client receives the reply.

    (d) a and b

    (e) a and c

16. Which of the following description(s) is true?

    (a) A transition's source state and target state cannot be the same state.

    (b) A guard is a condition that must be satisfied in order to enable an associated transition to fire.

    (c) A guard must be a Boolean expression.

    (d) b and c

    (e) a and c

17. Which of the following description(s) is true?

    (a) Each sequential process or thread can be modelled as an independent state machine.

    (b) A state machine can have multiple active states concurrently.

    (c) Each shared object in the system can be modelled as an independent state machine.

    (d) a and c

    (e) a, b, and c

18. Consider a *bank account* object, which maintains a balance and provides two operations:

```
void deposit( float amount );
void withdraw( float amount );
```

which increment and decrement the balance by the given amounts. Multiple clients might access these operations concurrently. Further, requests for withdrawal should not be allowed to cause the balance to become negative. Rather, such requests should block the client making the request, delaying the operation until the balance is sufficient to permit the withdrawal of the desired amount. Moreover, such blocking of one client on a withdrawl should not prevent another client from making a deposit.

19. One way to implement the behavior described in Question 18 is to implement the bank account as a monitor, which encapsulates the account balance and provides deposit and withdraw methods. Such a monitor should be implemented to avoid race conditions among concurrent clients, and withdrawals should not be allowed to cause the balance to become negative. As described in Question 18, such withdrawal requests should be delayed until the balance is sufficient to permit withdrawal of the desired amount. Below, please find a skeleton implementation for such a monitor. Please fill in the code for the deposit and withdraw methods.

```
class BankAccount{

   private:
        double balance;
        ACE_Thread_Mutex lock;   // the monitor lock
        ACE_Condition_Thread_Mutex  OKtoWithdraw; // condition variable


   public:

        BankAccount(): balance(0), lock(), OKtoWithdraw(lock) {}

        void deposit(double amount) {




        }

        void withdraw(double amount) {




        }

}
```

D.2 State machine modeling vs. non-modeling : Post-test questions

Evaluating UML 2.0 state diagrams: Post-test

November 29, 2007

Please answer these questions in order. Do not go back to prior questions.

Questions 1-3: **The readers-writers problem is a classic synchronization problem in which two distinct classes of threads, readers and writers, share access to a database. Multiple reader threads can be present in the database simultaneously. However, the writer threads must have exclusive access. That is, no other writer thread, nor any reader thread, may be present in the database while a given writer thread is present. Note: the reader thread must call startRead() to enter the database and it must call endRead() to exit the database. Similarly, the writer thread must call startWrite() to enter the database and it must call endWrite() to exit the database. Assume that state variables numReaders and numWriters are used to keep track of the number of client processes currently in the database.**

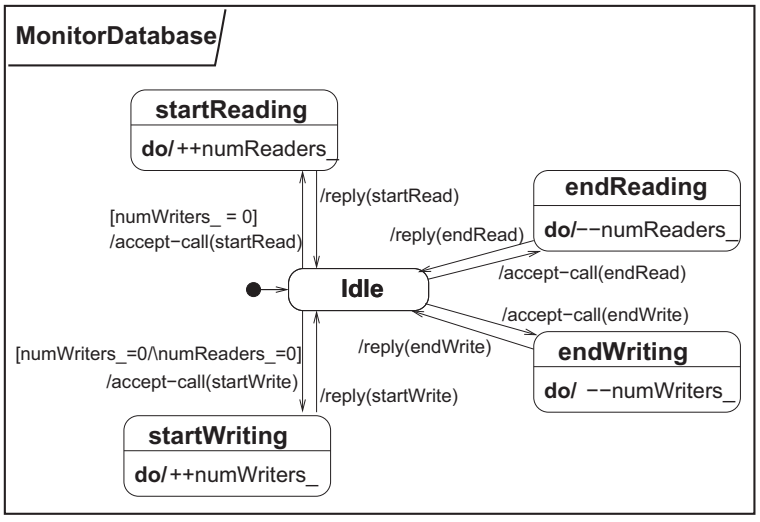


Figure 1: UML 2.0 state model of a MonitorDatabase object

1. See the state model for the shared database, as shown in figure 1. Mark each of the following statements as true or false.

(a) _____ A reader thread may enter the database when other reader threads are present.

(b) _____ A writer thread may enter the database when other writer threads are present.

(c) _____ Multiple reader threads may be in the startRead() method at the same time.

(d) _____ Multiple writer threads may be in the startWrite() method at the same time.

(e) _____ A reader has invoked startRead() and is reading the database. Before it invokes endRead(), a writer thread invokes startWrite() and a second reader thread invokes startRead(). The order of access by the writer thread and reader thread depends on thread scheduling.

(f) _____ In a monitor-based implementation of this shared database, when a reader thread invokes endRead(), it should call wait() on a condition variable.

(g) _____ In a monitor-based implementation of this shared database, when a reader thread invokes endRead(), it should call signal() to awaken a thread blocked on that condition variable.

(h) _____ In a monitor-based implementation of this shared database, when a reader thread invokes endRead(), it should call broadcast to awaken all threads blocked on that condition variable.

(i) _____ In a monitor-based implementation of this shared database, reader threads should acquire the lock only at the beginning of startRead() and release the lock only at the end of endRead().

```
// -*- C++ -*-

#ifndef MONITOR_DATABASE_H
#define MONITOR_DATABASE_H

#include <ace/Thread_Mutex.h>
#include <ace/Condition_Thread_Mutex.h>

const unsigned MAX_KEY = 10;

class MonitorDatabase
{
public:
  MonitorDatabase();

  void startRead();
  unsigned readData(unsigned key);
  void endRead();

  void startWrite();
  void writeData(unsigned key, unsigned value);
  void endWrite();

private:
  unsigned data_[MAX_KEY];

  // Number of readers reading in the database.
  unsigned int numReaders_;

  // Number of writers writing in the database.
  unsigned int numWriters_;

  // The monitor lock.
  ACE_Thread_Mutex lock_;

  // A condition variable synchronizing readers.
  ACE_Condition_Thread_Mutex okToRead_;

  // A condition variable synchronizing writers.
  ACE_Condition_Thread_Mutex okToWrite_;
};
```

Figure 2 - header file for a C++/ACE implementation of the MonitorDatabase.

```
void
MonitorDatabase::startWrite()
{
  lock_.acquire();

  while (numReaders_ > 0 || numWriters_ > 0) {
    okToWrite_.wait();
  }

  ++numWriters_;
}

...

void
MonitorDatabase::endWrite()
{
  --numWriters_;
  okToWrite_.signal();
  okToRead_.signal();
  lock_.release();
}
```

Figure 3 - sample implementations of the startWrite() and endWrite() methods for the MonitorDatabase.

2. **Figure 2 contains the header file for the monitor implementation of the shared database. Figure 3 contains sample implementations of the startWrite() and endWrite() methods of the monitor implementation. Identify any elements of these methods that are missing or incorrect, or write that they are correct. If any errors exist, explain their effects and show how to correct them.**

3. In the space below, write the C++/ACE code to implement the startRead() and endRead() methods for the MonitorDatabase depicted in the model and for which the header file appears in figure 2.

Questions 4-5: **Consider a variant of the "party" problem discussed in lecture, which we term the "matchmaker" problem. The matchmaking service maintains two variables: numGirls and numBoys. Whenever a girl "arrives" (as a result of a client invoking addGirl()), numGirls is incremented, after which the client may continue. Whenever a boy "arrives" (as a result of a client invoking addBoy()), numBoys is incremented, after which the client may continue. Whenever both a girl and a boy are present, a "matchmaker" client pairs them and both numGirls and numBoys are decremented.**

4. **Sketch a UML 2.0 state model of this shared matchmaking object.**

5. Write the code for the addBoy(), addGirl() and pair() methods of this shared matchmaking object. The header file for the implementation is seen below.

```
// -*- C++ -*-

#ifndef MATCHMAKER_H
#define MATCHMAKER_H

#include <ace/Thread_Mutex.h>
#include <ace/Condition_Thread_Mutex.h>

class MatchMaker
{
public:
  MatchMaker();

  void addBoy();
  void addGirl();
  void pair();
private:
  unsigned numBoys_;
  unsigned numGirls_;

  ACE_Thread_Mutex lock_;

  // A condition variable synchronizing boys.
  ACE_Condition_Thread_Mutex okToGoForBoys_;

  // A condition variable synchronizing girls.
  ACE_Condition_Thread_Mutex okToGoForGirls_;

  unsigned boyCounter_;
  unsigned girlCounter_;
};

#endif /* not MATCHMAKER_H */
```

Appendix E

Materials used in the state machine modeling vs. sequence diagram modeling study

E.1 State machine modeling vs. sequence diagram modeling : Pre-test questions

Evaluating UML 2.0 state diagrams: Pre-test

April 16, 2008

1. A *monitor* is:
   (a) an operation that must be performed entirely or not performed at all.
   (b) a sequential stream of execution.
   (c) a class-like programming language construct within which only one thread may execute concurrently.
   (d) a section of code that must be executed atomically.
   (e) a section of code in which the result of computation depends on the ordering of thread execution.

2. A race condition is:
   (a) the situation in which multiple threads or processes read and write a shared data item and the final result depends on the order of execution.
   (b) the situation where no progress can be made because each thread or process is waiting for an event that only another process or thread in the set can cause.
   (c) the condition in which only one thread at a time may access a shared resource.
   (d) an operation that must be performed entirely or not performed at all.
   (e) a section of code that must be executed atomically.

3. A critical section is:
   (a) a section of code that may be accessed by only one thread at a time.
   (b) a sequential stream of execution.
   (c) a variable used to synchronize threads.
   (d) an operation that suspends/blocks execution of the calling process if a certain condition is true.
   (e) an operation that resumes the execution of some suspended thread or process by placing it in the processor ready queue.

4. Which of the following is true of the typical use of condition variables in a monitor?

   (a) Invoking wait() on a condition variable suspends the invoking thread.

   (b) Invoking signal() on a condition variable resumes a thread waiting on that condition variable.

   (c) If there is no waiting thread, a signal can be stored/delayed to resume the next thread that waits on that condition variable.

   (d) a and b only

   (e) a, b, and c

5. Which of the following description(s) is true?

   (a) Acquiring a mutex lock takes place before anything else in a monitor method.

   (b) Releasing a mutex lock takes place after everything else in a monitor method.

   (c) The monitor mutex lock must be both acquired and released in a monitor method to ensure correct behavior.

   (d) a and b only

   (e) a, b, and c

6. What feature of the monitor implementation prevents race conditions in updating counting variables?

   (a) The evaluation of a condition.

   (b) The wait() operation on condition variables.

   (c) The lock on the monitor.

   (d) The signal() operation on condition variables.

   (e) That a thread releases the lock prior to suspending in the wait() operation.

7. In a monitor implementation, the monitor lock serves the following purpose:

   (a) To prevent race conditions among threads which are sharing access to the monitor.

   (b) To prevent deadlock among threads which are sharing access to the monitor.

   (c) To ensure fairness among threads which are sharing access to the monitor.

   (d) All of the above.

   (e) None of the above.

8. In the implementation of a monitor that employs condition synchronization, which of the following features contributes to deadlock avoidance among threads that share the monitor?

   (a) Threads resumed by `signal()` reacquire the monitor lock prior to returning from `wait()`.

   (b) Threads release the monitor lock prior to suspending in the `wait()` operation.

   (c) Threads acquire the monitor lock upon entry into a method of the monitor.

   (d) A thread blocks while trying to acquire the monitor lock when another thread holds the lock.

   (e) None of the above.

9. Which of the following does a monitor guarantee?

   (a) Once a thread T enters the monitor, no other thread may execute any methods within the monitor until T returns from that method.

   (b) Only one thread at a time may execute within the monitor.

   (c) Multiple threads may enter a monitor concurrently provided that none of these threads modifies any of the data associated with the monitor object.

   (d) a and c only

   (e) a and b only

Questions 10 through 14 refer to a UML 2.0 state model of a BoundedBuffer monitor object, as depicted in Figure 2.

10. In the diagram of Figure 2, the label "[q.size < MAX]" is known as a(n):

    (a) action

    (b) activity

    (c) call

    (d) guard

    (e) state label

11. The BoundedBuffer enters the "Idle" state

    (a) upon creation

    (b) upon exiting the Pushing state

    (c) upon exiting the Pulling state

    (d) b and c

    (e) a, b, and c

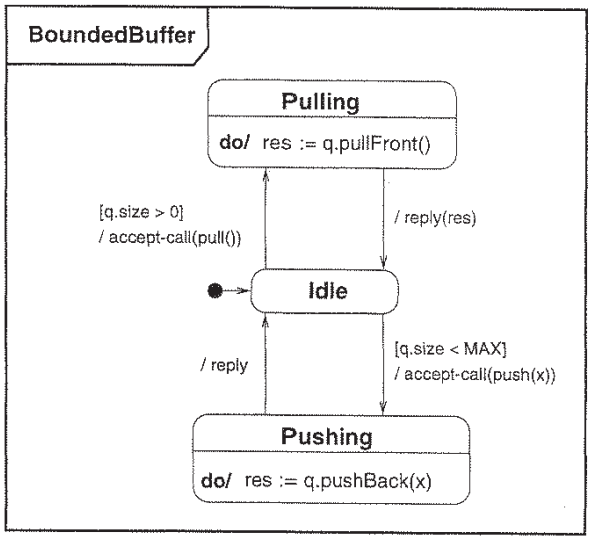


Figure 1: UML 2.0 state model of a Bounded Buffer monitor object

12. When a client process calls pull() on an instance of the BoundedBuffer process,

    (a) the BoundedBuffer process stops looping on q.pullFront(), exits the Pulling state, and then replies with the result of q.pullFront().

    (b) If (q.size > 0), the BoundedBuffer process accepts the pull() call and enters the Pulling state, where it loops on q.pullFront() until a response is generated. It then replies to the calling process with that response.

    (c) If (q.size > 0), the BoundedBuffer process accepts the pull() call and enters the Pulling state, where it executes and obtains a reply from q.pullFront(). It then replies to the calling process with that response.

    (d) The BoundedBuffer process accepts the call to pull() and then checks the condition (q.size > 0). If the condition is true, the BoundedBuffer process enters the Pulling state, where it loops on q.pullFront() until a response is generated. It then replies to the calling process with that response.

    (e) The BoundedBuffer process accepts the call to pull() and then checks the condition (q.size > 0). If the condition is true, the BoundedBuffer process enters the Pulling state, where it executes and obtains a reply from q.pullFront(). It then replies to the calling process with that response.

13. Assume that this BoundedBuffer is shared by multiple client processes, that MAX is 10 and the current queue size is 1. What will happen if three client processes invoke push(x) calls on the BoundedBuffer at virtually the same time?

    (a) The BoundedBuffer process will enter the Pushing state with a count of 3.

    (b) One client process will enter the Pushing state and two client processs will remain in the Idle state.

    (c) The BoundedBuffer process will accept the push(x) call of the first client process and enter the Pushing state. The other calls will be queued up until the pushBack(x) call is performed.

    (d) The BoundedBuffer process will accept the push(x) call of the first client process and enter the Pushing state. The other calls will be queued up until the BoundedBuffer has issued a reply and entered the Idle state.

    (e) The BoundedBuffer will accept the push(x) call of the first client process and enter the Pushing state. The other calls will be queued up until the BoundedBuffer has accepted a call to pull(), entered the Pulling state, and then issued a reply and entered the Idle state.

14. Assume that this BoundedBuffer is shared by multiple threads, that MAX is 10 and the current queue size is 10. What will happen if a client process invokes a push(x) call on the BoundedBuffer?

    (a) The BoundedBuffer will reject the push(x) call.

    (b) The BoundedBuffer will queue up the push(x) call until (q.size < MAX), thus blocking the client process.

    (c) The BoundedBuffer will queue up the push(x) call until (q.size < MAX). The client process may continue.

    (d) The BoundedBuffer will accept the push(x) call and then evaluate the condition (q.size < MAX). The client process will block.

    (e) The BoundedBuffer will accept the push(x) call and then evaluate the condition (q.size < MAX). The client process will continue.

15. In a rendezvous style communication, which of the following description(s) is true?

    (a) After sending a request to a server process, a client process blocks until the server replies.

    (b) Upon completion of serving a request from a client process, a server process sends a reply to the client process.

    (c) When replying to a request from a client process, a server process blocks until the client receives the reply.

    (d) a and b

    (e) a and c

16. Which of the following description(s) is true?

    (a) A transition's source state and target state cannot be the same state.

    (b) A guard is a condition that must be satisfied in order to enable an associated transition to fire.

    (c) A guard must be a Boolean expression.

    (d) b and c

    (e) a and c

17. Which of the following description(s) is true?

    (a) Each sequential process or thread can be modelled as an independent state machine.

    (b) A state machine can have multiple active states concurrently.

    (c) Each shared object in the system can be modelled as an independent state machine.

    (d) a and c

    (e) a, b, and c

Questions 18 - 21 refer to the attached BoundedBuffer sequence diagram.

18. Which of the following statement(s) is true?

    (a) p and c are passive objects
    (b) p, b, q and c are passive objects
    (c) p is the only active object
    (d) b and q are active objects
    (e) p and c are active objects

19. Which of the following statement(s) is true?

    (a) the diagram depicts only one active thread, hosted by p
    (b) the diagram depicts only one active thread, hosted by c
    (c) the diagram depicts two active threads, hosted by b and q
    (d) the diagram depicts two active threads, hosted by p and c
    (e) the diagram depicts four active threads, hosted by p, b, q, and c

20. Which of the following statement(s) is true?

    (a) the pull method is hosted by object c
    (b) the pushBack method is hosted by object b
    (c) the push method is hosted by object p
    (d) the pullFront method is hosted by object b
    (e) none of the above are correct

21. Assume, at the beginning of the interaction depicted in the Figure, that the queue q contains y. What happens as a result of this interaction?

    (a) c gets a copy of x; q contains y
    (b) c gets a copy of y; q contains x
    (c) c gets a copy of x; q contains x and y
    (d) c gets a copy of y; q contains x and y

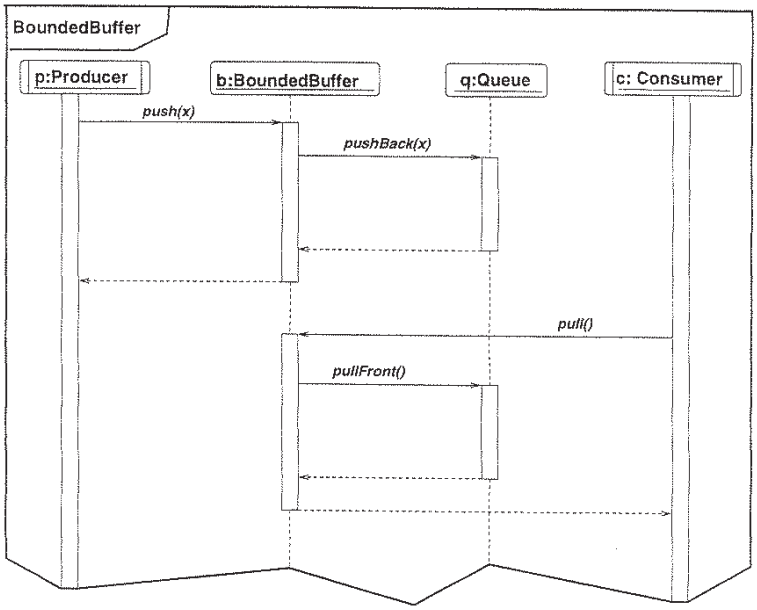


Figure 2: UML 2.0 state model of a Bounded Buffer monitor object

E.2 State machine modeling vs. sequence diagram modeling : Post-test questions

Diagrams for Concurrency: Post-Test

ID Number: _____

Figure 1 depicts the C++ declaration of a class `RWDatabase`, whose instances are databases that permit concurrent access by multiple threads according to the *readers-writers* synchronization policy. This policy allows multiple threads to simultaneously access a shared resource (the database) provided these accesses do not modify the resource. The policy recognizes two distinct types of threads: *readers*, which only read data, and *writers*, which may write into the database. Because readers only read data, it is safe for multiple readers to access the database simultaneously. However, a writer must access the database exclusively—that is, no threads of either type can access the database simultaneously with a writer.

This policy is implemented using a protocol by which readers and writers request authorization prior to accessing any data and relinquish authorization when they are done with the data. A reader obtains and relinquishes authorization by invoking `startRead` and `endRead`, respectively. The methods `startRead` and `endRead` contain *critical sections* and are implemented as *monitor methods*. Once authorized, actual reading is accomplished by invoking the `read` method. The `read` method is a *non-monitor method*.

Similarly, a writer obtains and relinquishes authorization by invoking `startWrite` and `endWrite`, respectively. The methods `startWrite` and `endWrite` contain *critical sections* and are implemented as *monitor methods*. Actual writing of data is accomplished by invoking the `write` method. The `write` method is a *non-monitor method*.

You may assume that all threads adhere to these protocols when accessing the databases.

To implement this policy, class `RWDatabase` uses:

- a mutex lock, `lock`,
- two counter variables, `numReaders` and `numWriters`, which are used to track the number of client threads of each type that are currently authorized to access (i.e., read or write) the database, and
- two condition variables, `okToRead` and `okToWrite`.

The attached figures represent a header file for a C++/ACE implementation of the database, and UML 2.0 sequence diagrams of the executions of a system containing reader clients (0-2) , writer clients (0-2), and an instance of `RWDatabase`. Please refer to these figures in answering the questions that follow.

1. Assume that only two threads exist in this system: *reader1* and *reader2*. At the start of the program, reader thread *reader2* is running and has just invoked the `startRead` method and then the `read` method. The invocation of `read` has not yet returned. A context switch occurs and the *reader1* thread begins to run.

   Which of the following event sequences could happen next? **Circle YES if the event sequence is possible; otherwise, circle NO.**

   (a) *reader1* invokes `startRead` and then blocks on the monitor lock.

   YES NO

   (b) *reader1* invokes `startRead`, followed by `read`, followed by `endRead`.

   YES NO

   (c) *reader1* invokes `read`, followed by `startRead`, followed by `endRead`.

   YES NO

   (d) *reader1* invokes `startRead` and then blocks because *reader2* has been granted read authorization.

   YES NO

   (e) *reader1* invokes `startRead` and is then context-switched out before the call returns. *reader2* completes its invocation of `read`.

   YES NO

2. Assume that only two threads exist in this system: *writer1* and *reader1*.

   Reader thread *reader1* is running and has just invoked the `startRead` method, followed by `read`, followed by `endRead`. The `endRead` method has returned. A context switch occurs and writer thread *writer1* begins running and has invoked the `startWrite` method. The `startWrite` method has returned. Another context switch occurs and the *reader1* thread begins to run.

   Which of the following event sequences could happen next? **Circle YES if the event sequence is possible; otherwise, circle NO.**

   (a) *reader1* invokes `startRead` and then blocks on the monitor lock.

   YES NO

   (b) *reader1* invokes `startRead`, followed by `read`, followed by `endRead`.

   YES NO

   (c) *reader1* invokes `startRead` and then blocks because *writer1* has been given write authorization.

   YES NO

   (d) *reader1* invokes `startRead` and then blocks because *writer1* has been given write authorization. The writer thread then invokes `write`, followed by `endWrite`, followed by `startWrite`.

   YES NO

   (e) *reader1* invokes `startRead` and then blocks because *writer1* has been given write authorization. *writer1* then invokes `write`, followed by `endWrite`. When the call to `endWrite` returns, *reader1* is in the ready state.

   YES NO

3. Assume that only two threads exist in this system: *reader1* and *writer1*. At the start of the program writer thread *writer1* is running and has invoked the `startWrite` method, followed by the `write` method, followed by the `endWrite` method. The `endWrite` method has returned. A context switch occurs and Reader thread *reader1* is now running and has invoked the `startRead` method, followed by the `read` method. The `read` method call has not yet returned. Another context switch occurs and the *writer1* thread begins to run.

   Which of the following event sequences could happen next? **Circle YES if the event sequence is possible; otherwise, circle NO.**

   (a) *writer1* invokes `startWrite` and then blocks on the monitor lock.

   YES NO

   (b) *writer1* invokes `startWrite`, followed by `write`, followed by `endWrite`.

   YES NO

   (c) *writer1* invokes `startWrite` and then blocks because the reader has been granted read authorization.

   YES NO

   (d) *writer1* invokes `startWrite` and then blocks because the reader has been granted read authorization. *reader1* then completes the call to `read`, followed by `endRead`. The reader thread continues to run.

   YES NO

   (e) The writer invokes `startWrite` and then blocks because the reader has been granted read authorization. The reader thread then completes the call to `read`, followed by `endRead`. When the call to `endRead` returns, *writer1* is in the ready state.

   YES NO

4. Assume that only two threads exist in this system: *writer1* and *writer2*. At the start of the program writer thread *writer2* is running and has invoked the `startWrite` method, followed by the `write` method. The `write` method call has returned. A context switch occurs and the *writer1* thread begins to run.

   Which of the following event sequences could happen next? **Circle YES if the event sequence is possible; otherwise, circle NO.**

   (a) *writer1* invokes `startWrite` and then blocks on the monitor lock.

   YES NO

   (b) *writer1* invokes `startWrite`, followed by `write`, followed by `endWrite`.

   YES NO

   (c) *writer1* invokes `startWrite` and then blocks because *writer2* has write authorization.

   YES NO

   (d) *writer1* invokes `startWrite` and then blocks because *writer2* has write authorization. *writer2* calls `endWrite`, then `startWrite`, and then `write`.

   YES NO

   (e) *writer1* invokes `startWrite` and then blocks because *writer2* has write authorization. *writer2* calls `endWrite`. When the call to `endWrite` returns, *writer1* is in the ready state.

   YES NO

5. Assume that only three threads exist in this system: *reader1*, *writer1*, and *reader2*. At the start of the program, reader thread *reader2* is running and has invoked the `startRead` method. The call to `startRead` has returned. A context switch occurs. Thread *writer1* begins running and invokes the `startWrite` method.

   Which of the following event sequences could happen next? **Circle YES if the event sequence is possible; otherwise, circle NO.**

   (a) *writer1* blocks on the monitor lock.

   YES NO

   (b) *writer1* blocks because *reader2* has been granted read authorization.

   YES NO

   (c) *writer1* blocks because *reader2* has been granted read authorization. A context switch occurs. *reader1* begins to run.

   YES NO

   (d) *writer1* blocks because *reader2* has been granted read authorization. A context switch occurs. *reader1* begins to run and invokes a `startRead` method.

   YES NO

   (e) *writer1* blocks because *reader2* has been granted read authorization. A context switch occurs. *reader1* begins to run and invokes a `startRead` method and then a `read` method.

   YES NO

   (f) *writer1* blocks because *reader2* has been granted read authorization. A context switch occurs. *reader1* begins to run and invokes a `startRead` method. *reader1* blocks because *writer1* has been granted write authorization.

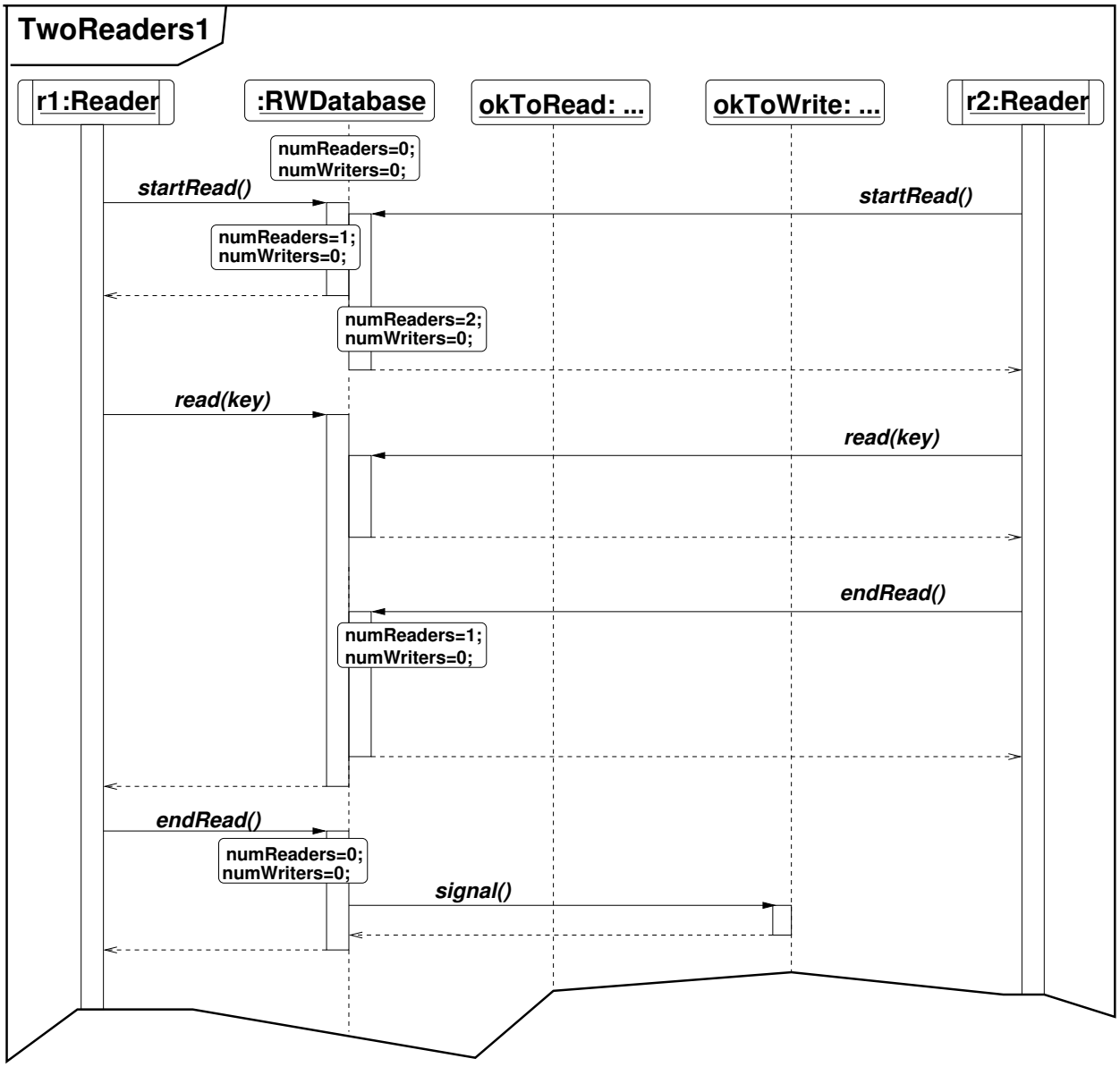
   YES NO

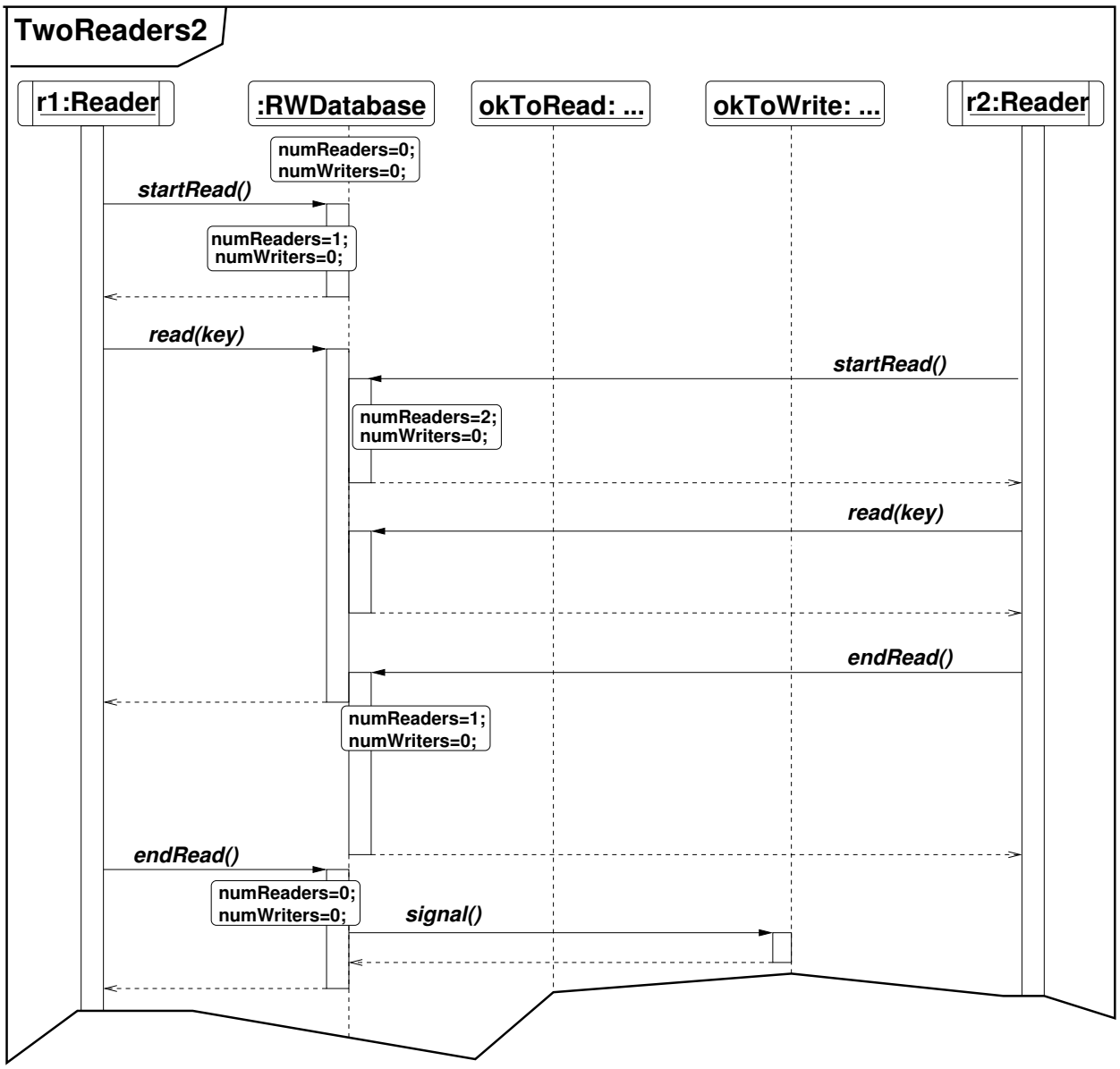
6. In the space below, implement methods for `startRead`, `endRead`, `startWrite`, and `endWrite`.

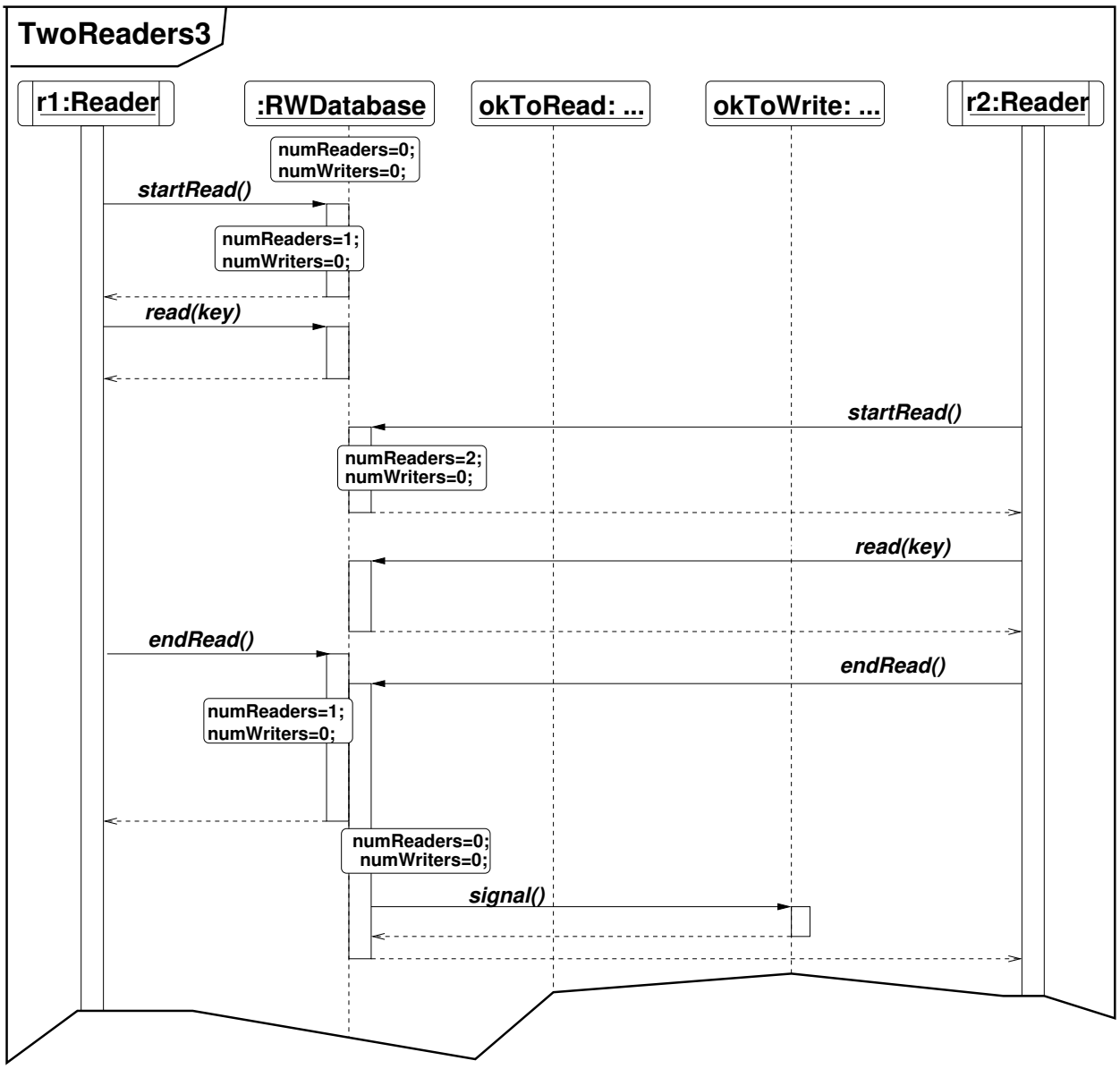
```
const unsigned MAX_KEY = 10;

class RWDatabase
{
public:
  RWDatabase();

  void startRead();            // a monitor method
  unsigned read(unsigned key); // a non-monitor method
  void endRead();              // a monitor method

  void startWrite();                          // a monitor method
  void write(unsigned key, unsigned value); // a non-monitor method
  void endWrite();                            // a monitor method

private:
  unsigned data[MAX_KEY];

  // Number of threads currently authorized to read.
  unsigned int numReaders;

  // Number of threads currently authorized to write.
  unsigned int numWriters;

  // The monitor lock.
  ACE_Thread_Mutex lock;

  // A condition variable used to signal readers.
  ACE_Condition_Thread_Mutex okToRead;

  // A condition variable used to signal writers.
  ACE_Condition_Thread_Mutex okToWrite;
};
```

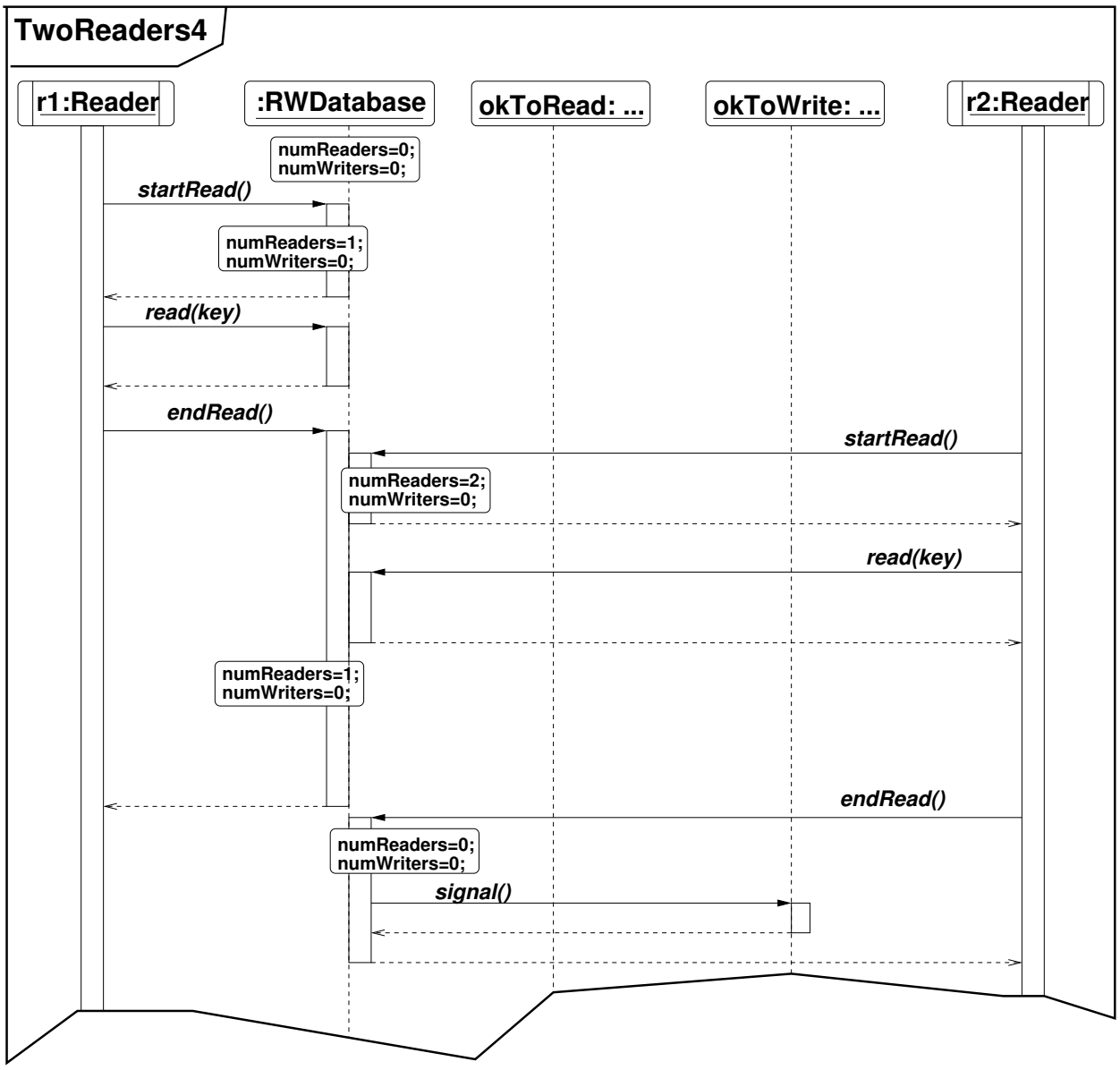
Figure 1: A C++/ACE implementation of the database with method definitions elided.

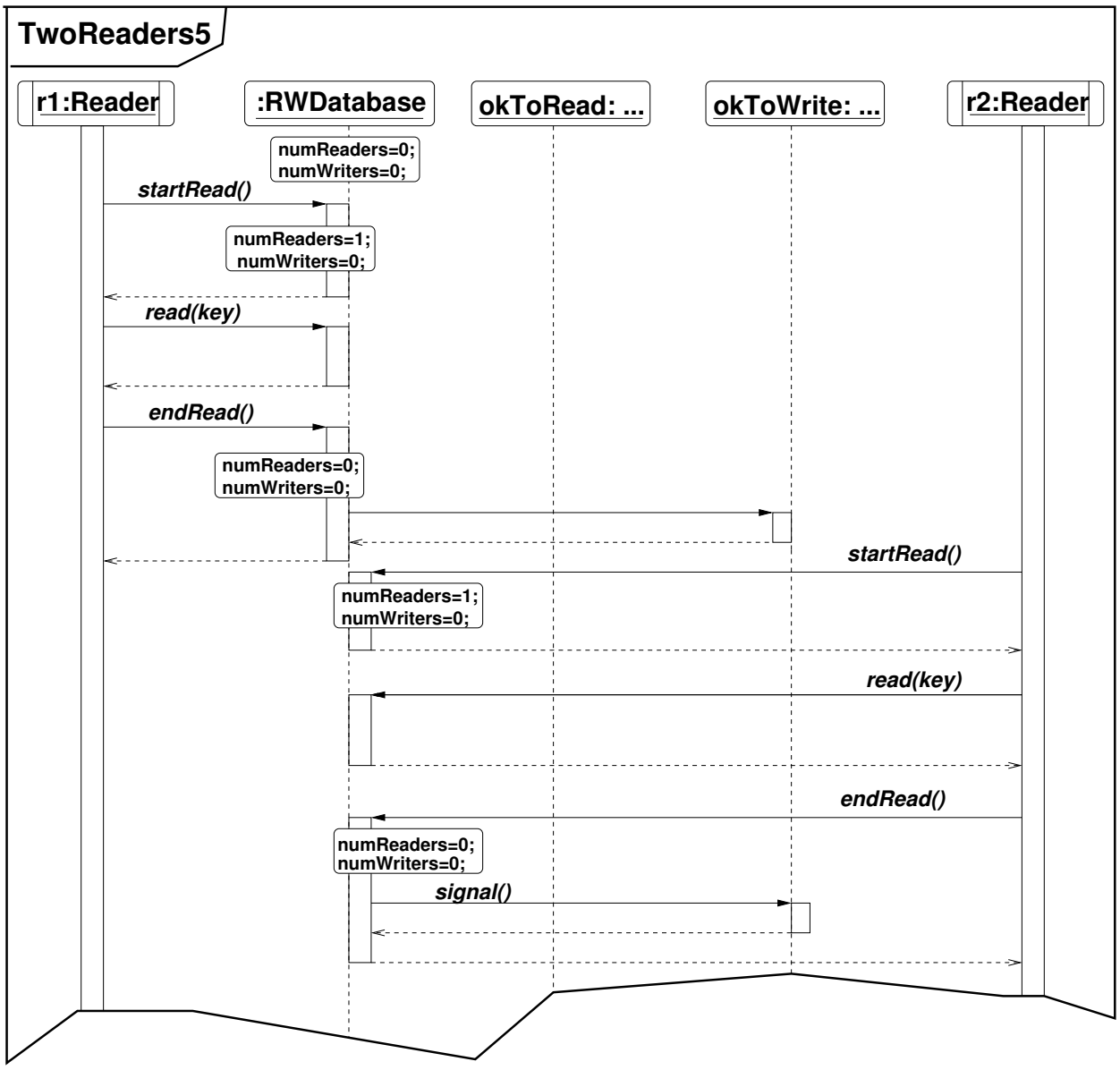
E.3 The nineteen UML sequence diagrams modeling the monitor-based readers-writer implementation

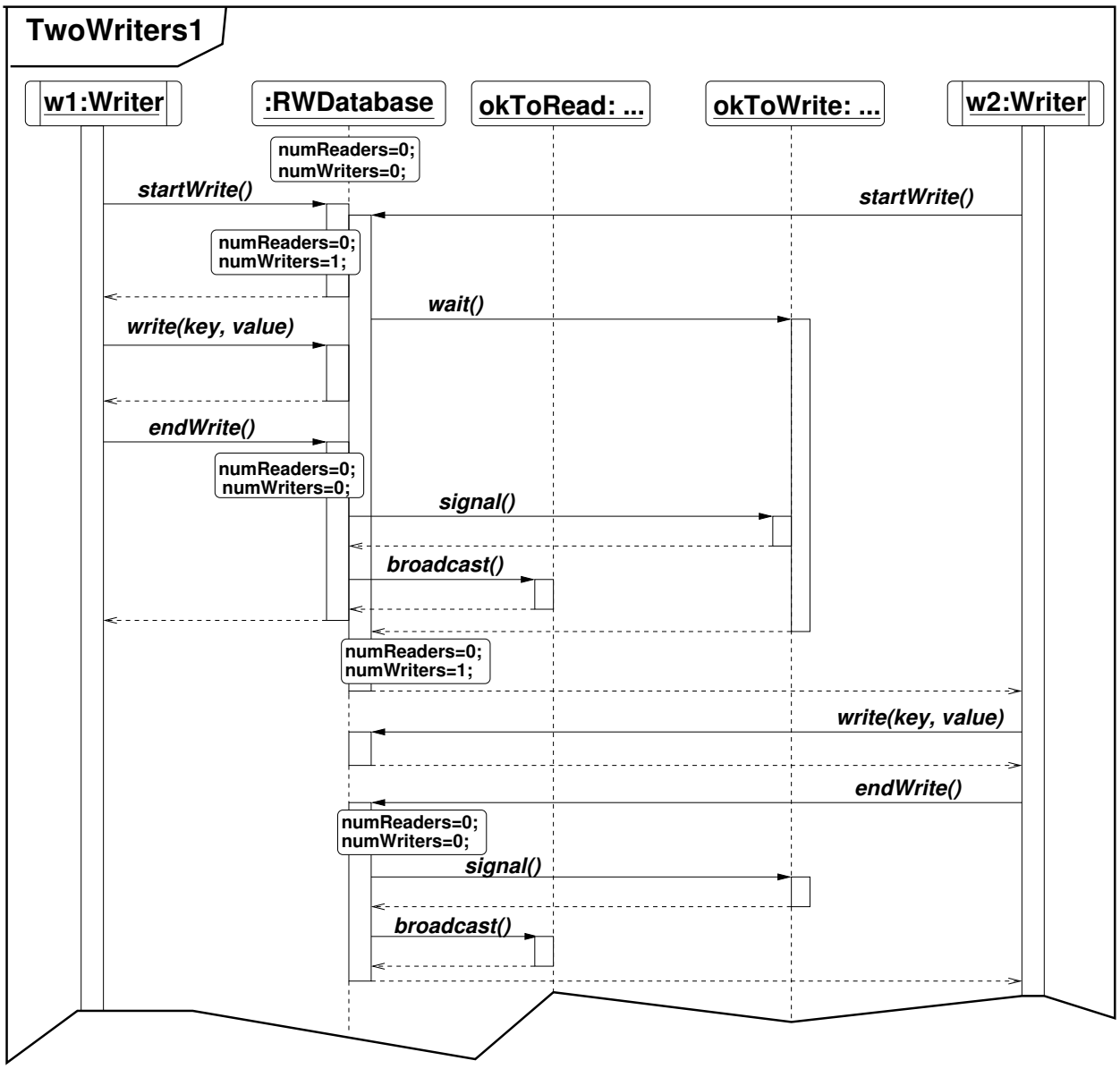


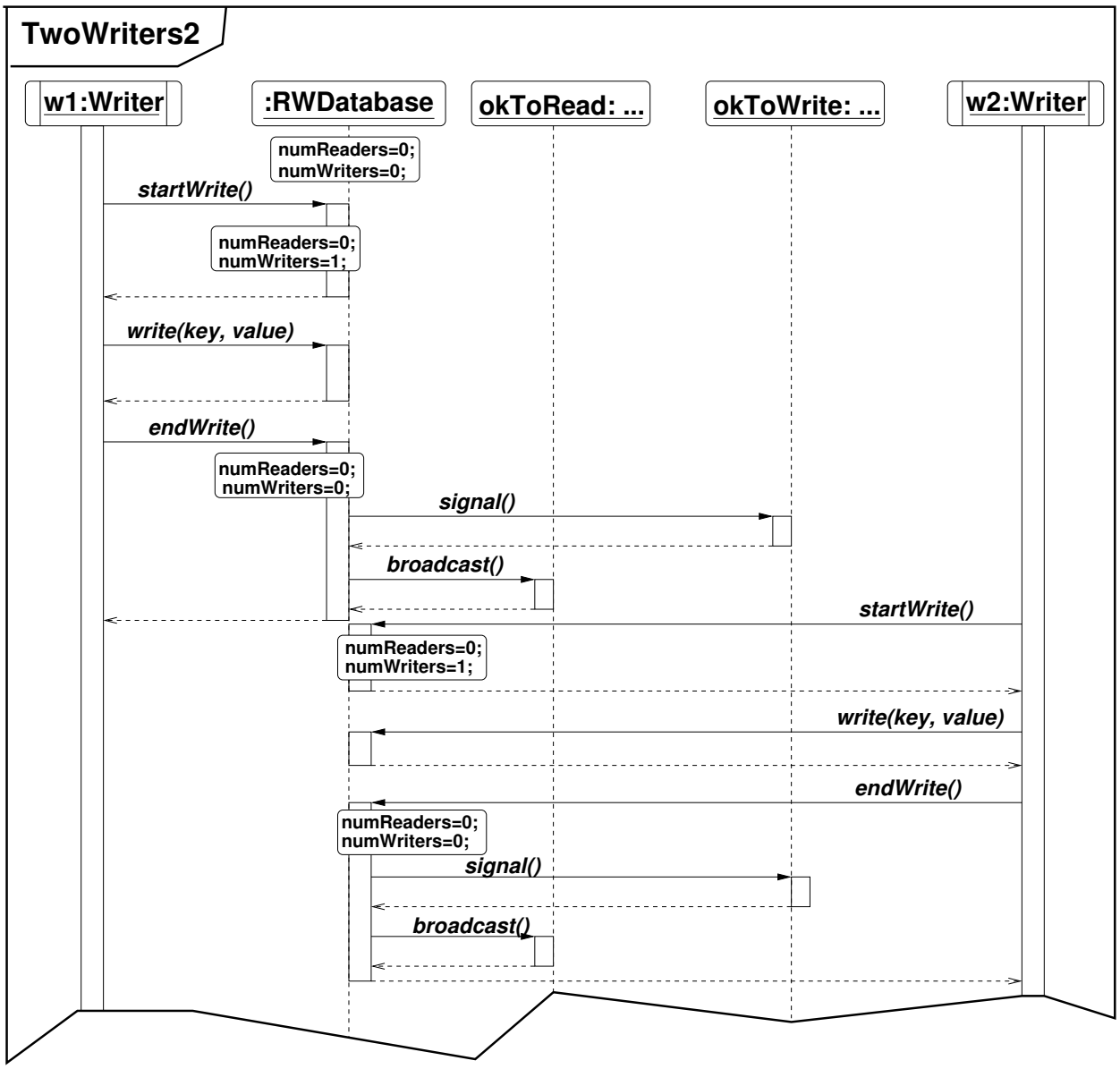

TwoReaders2
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
r2:Reader
numReaders=0;
numWriters=0;
startRead()
numReaders=1;
numWriters=0;
read(key)
startRead()
numReaders=2;
numWriters=0;
read(key)
endRead()
numReaders=1;
numWriters=0;
endRead()
numReaders=0;
numWriters=0;
signal()

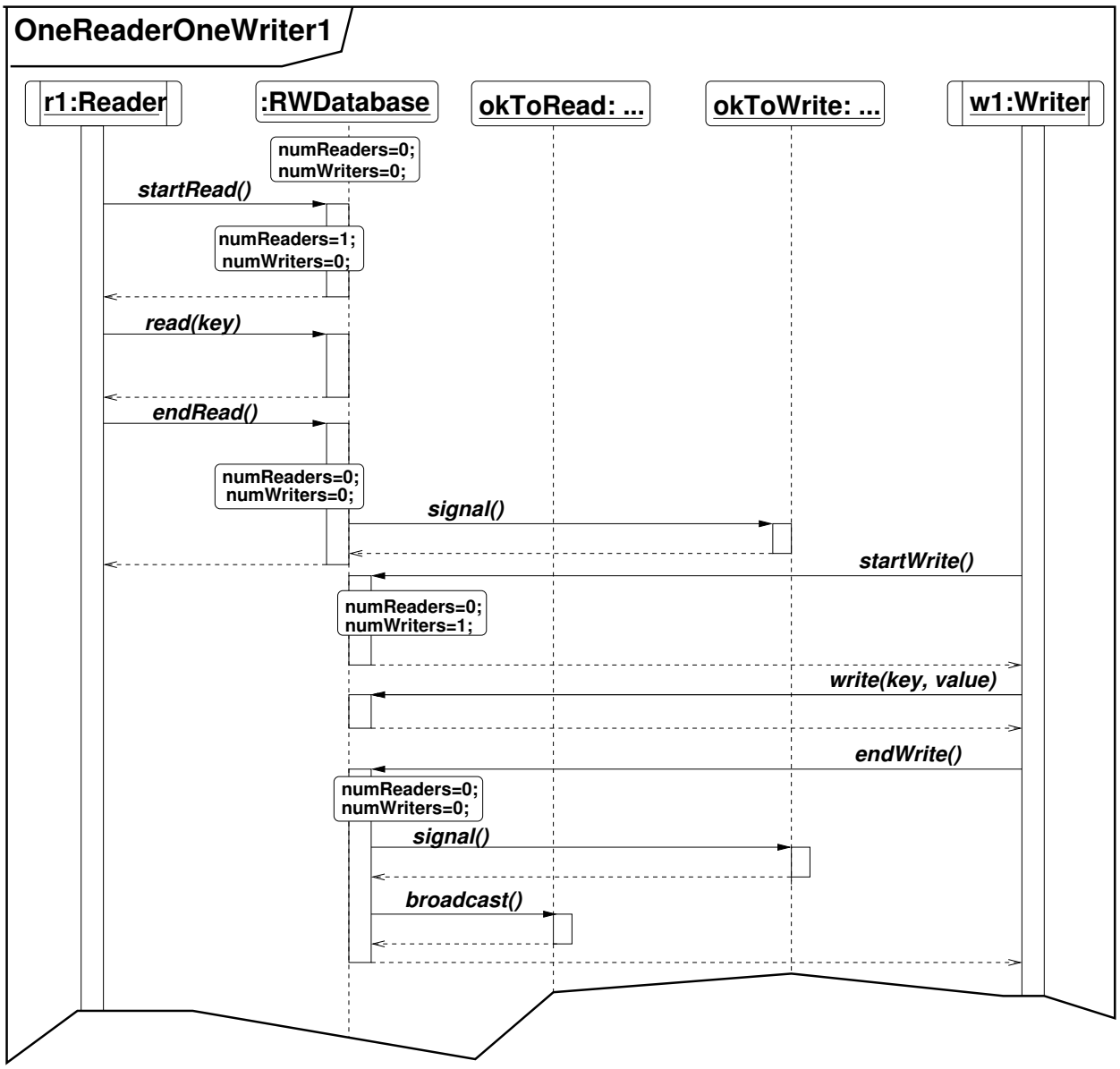

TwoReaders3
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
r2:Reader
numReaders=0;
numWriters=0;
startRead()
numReaders=1;
numWriters=0;
read(key)
startRead()
numReaders=2;
numWriters=0;
read(key)
endRead()
endRead()
numReaders=1;
numWriters=0;
numReaders=0;
numWriters=0;
signal()

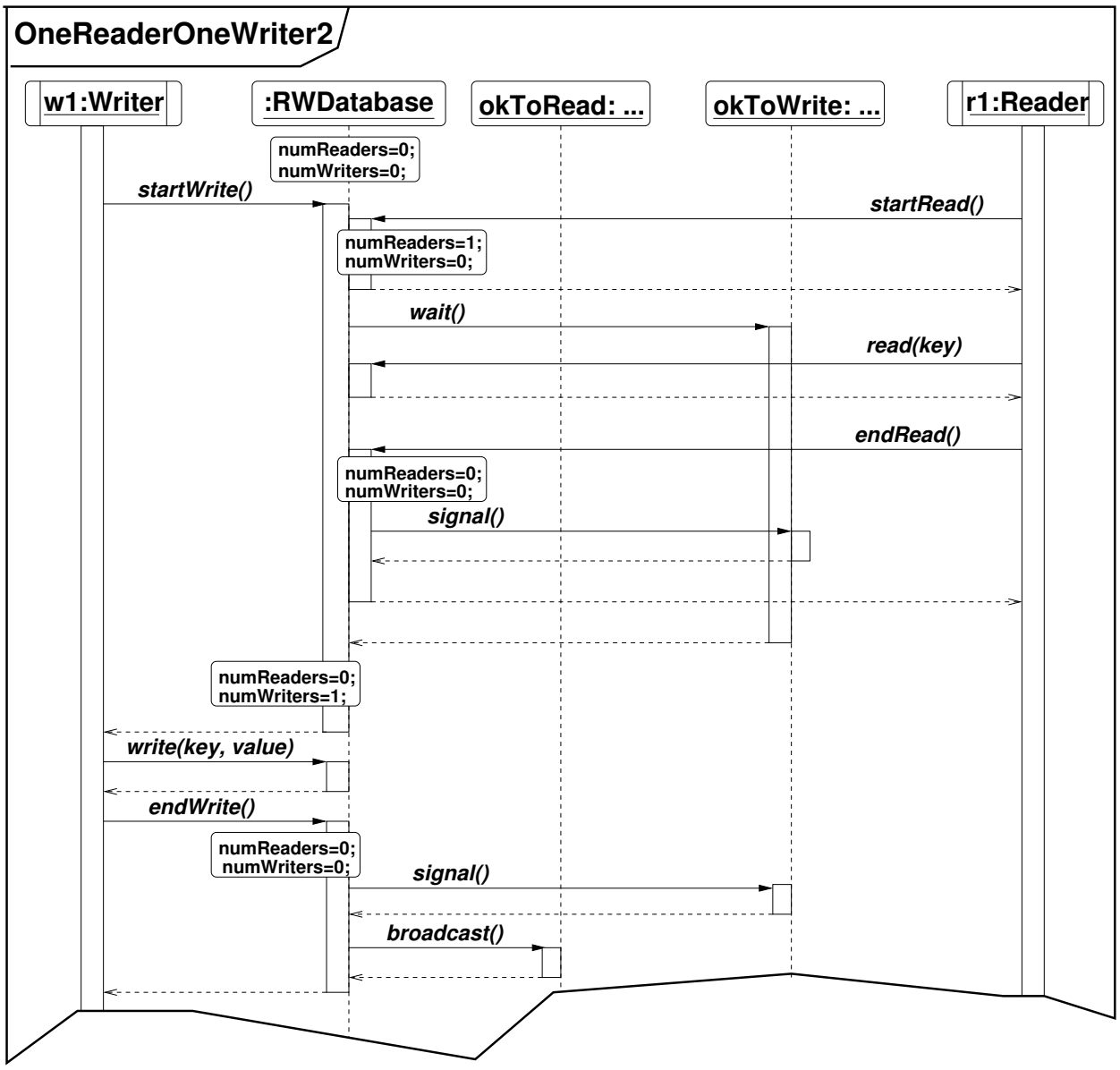

TwoReaders4
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
r2:Reader
numReaders=0;
numWriters=0;
startRead()
numReaders=1;
numWriters=0;
read(key)
endRead()
startRead()
numReaders=2;
numWriters=0;
read(key)
numReaders=1;
numWriters=0;
endRead()
numReaders=0;
numWriters=0;
signal()

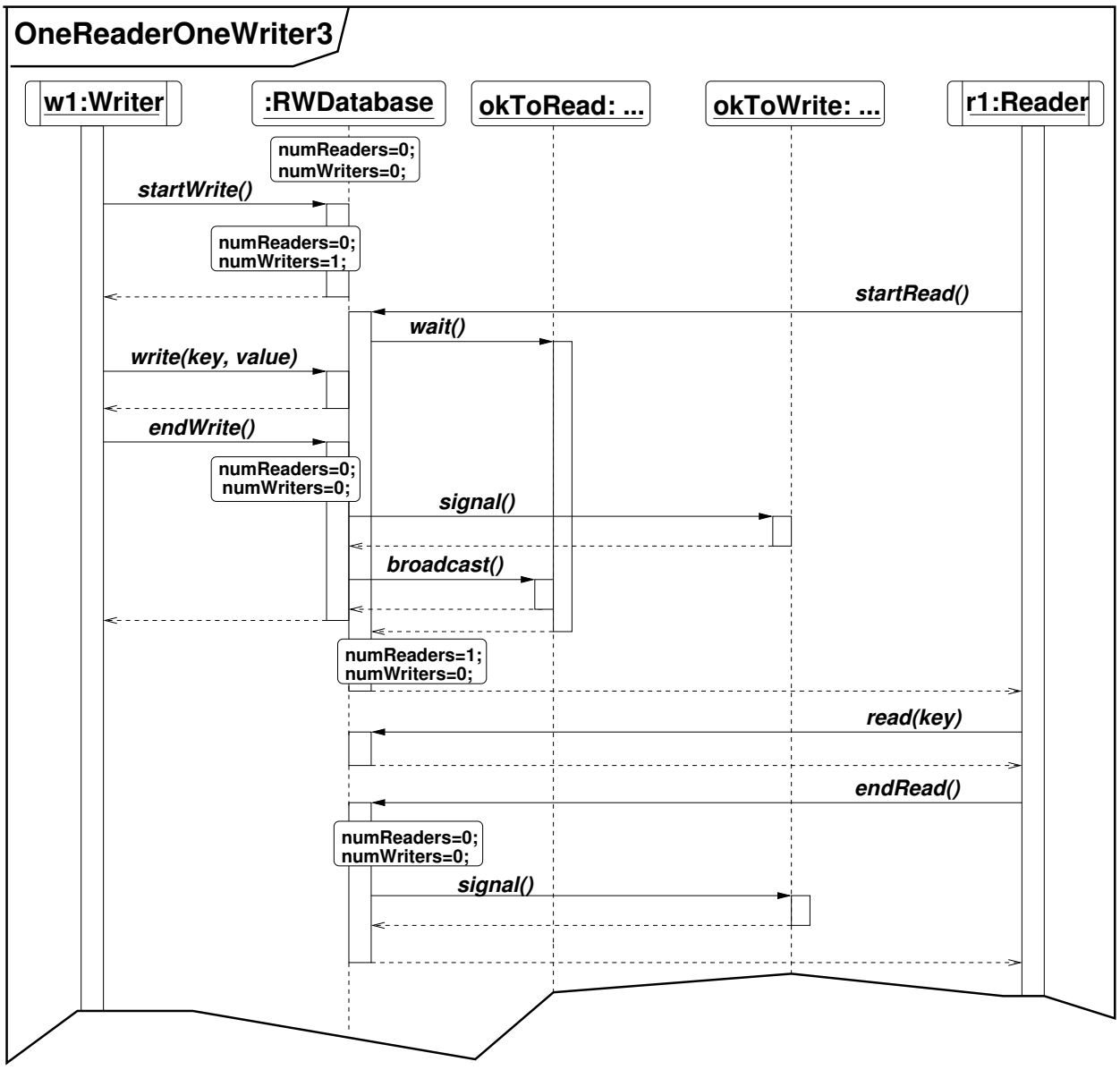

TwoReaders5
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
r2:Reader
numReaders=0;
numWriters=0;
startRead()
numReaders=1;
numWriters=0;
read(key)
endRead()
numReaders=0;
numWriters=0;
startRead()
numReaders=1;
numWriters=0;
read(key)
endRead()
numReaders=0;
numWriters=0;
signal()

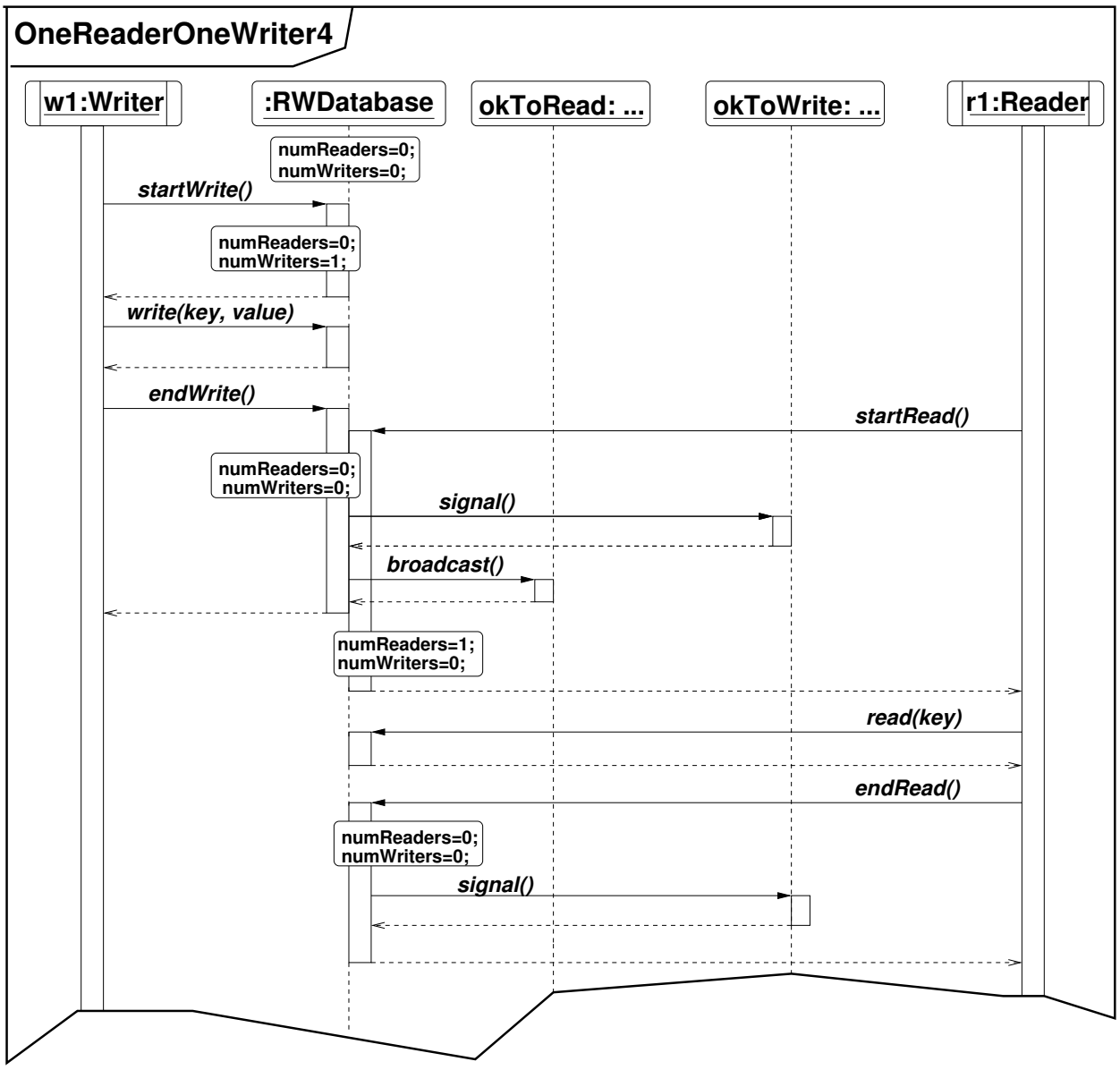

TwoWriters1
w1:Writer
:RWDatabase
okToRead: ...
okToWrite: ...
w2:Writer
numReaders=0;
numWriters=0;
startWrite()
startWrite()
numReaders=0;
numWriters=1;
wait()
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()

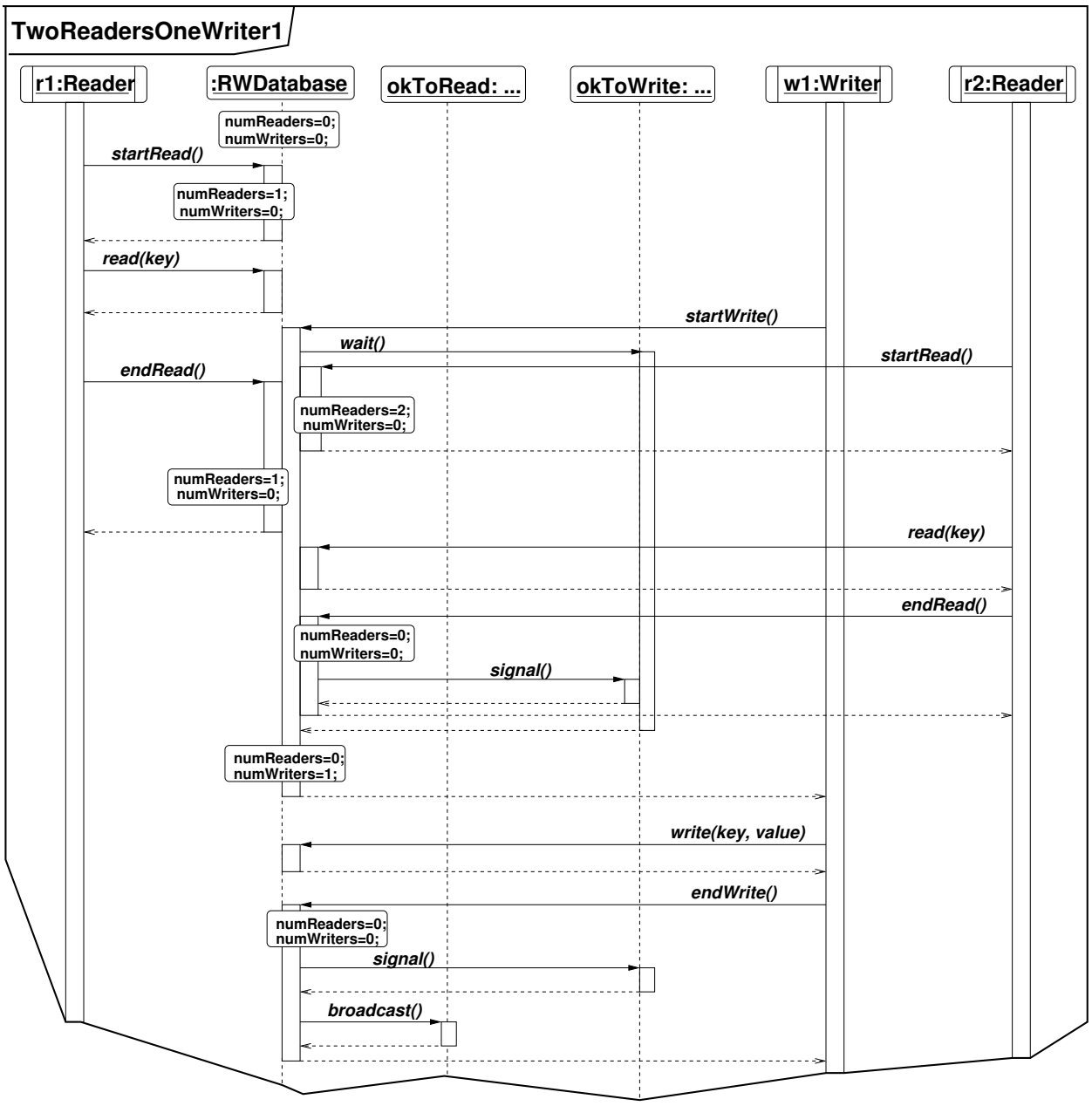

TwoWriters2
w1:Writer
:RWDatabase
okToRead: ...
okToWrite: ...
w2:Writer
numReaders=0;
numWriters=0;
startWrite()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()
startWrite()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()

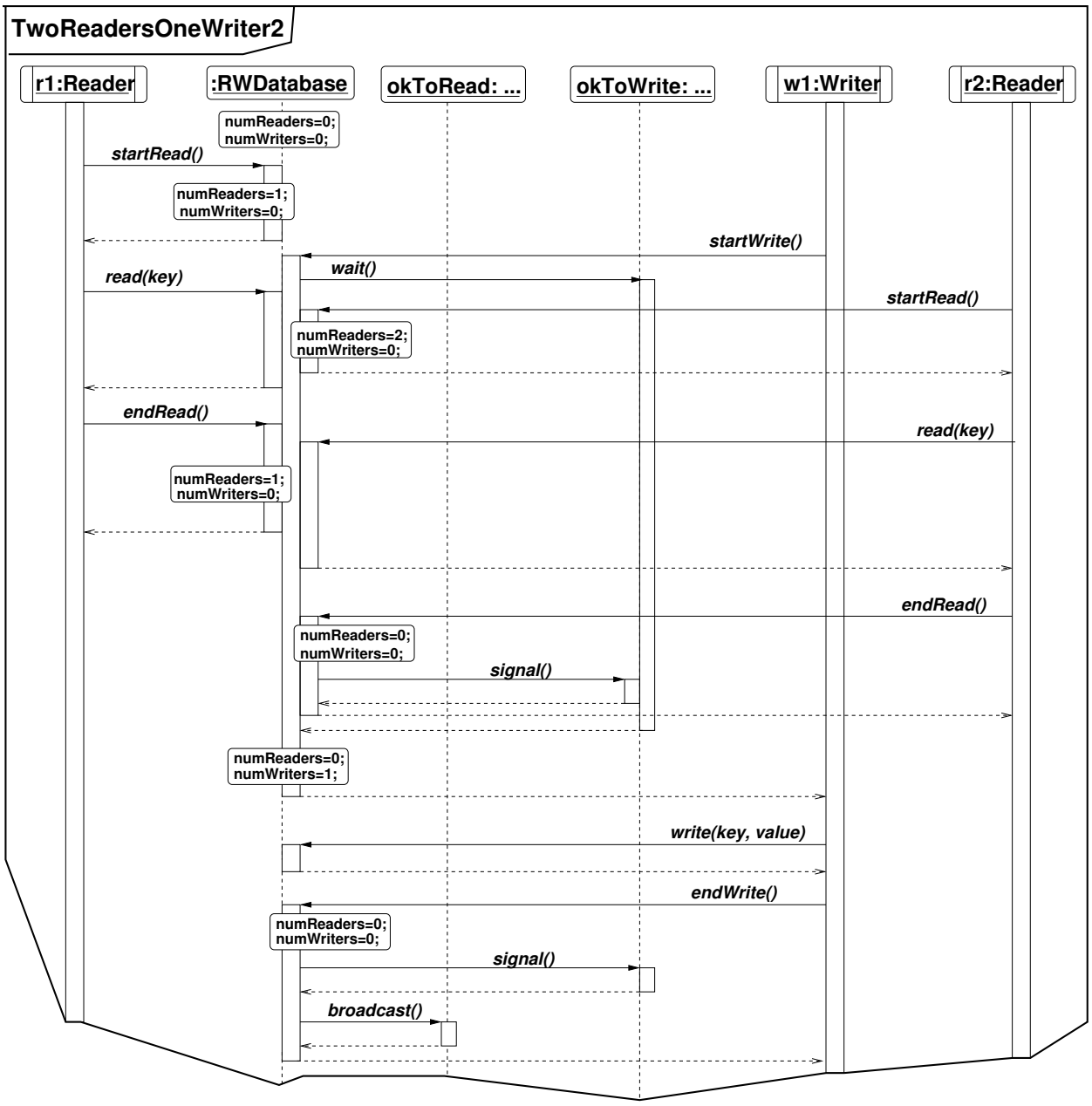

OneReaderOneWriter1
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
w1:Writer
numReaders=0;
numWriters=0;
startRead()
numReaders=1;
numWriters=0;
read(key)
endRead()
numReaders=0;
numWriters=0;
signal()
startWrite()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()

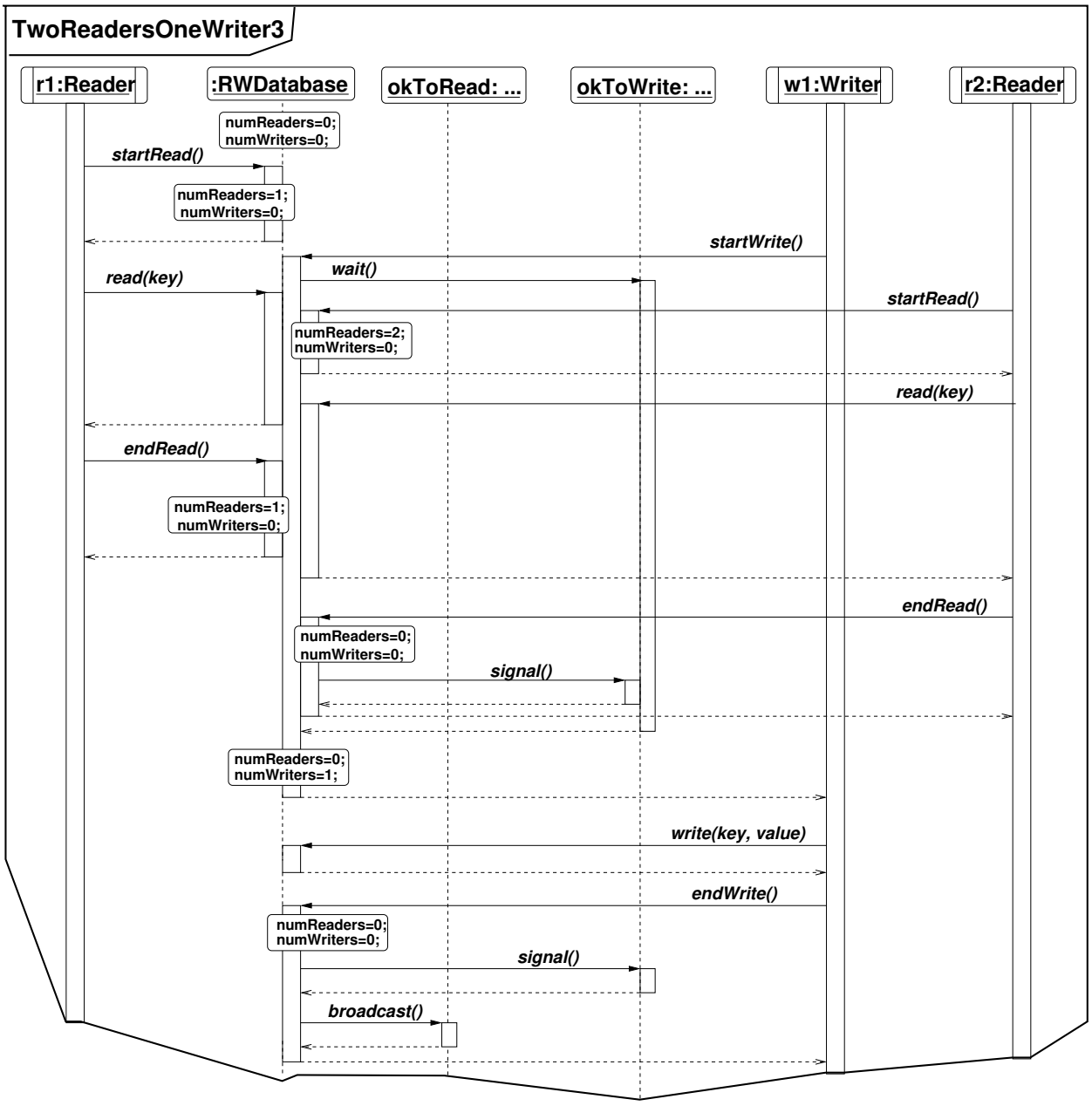

OneReaderOneWriter2
w1:Writer
:RWDatabase
okToRead: ...
okToWrite: ...
r1:Reader
numReaders=0;
numWriters=0;
startWrite()
startRead()
numReaders=1;
numWriters=0;
wait()
read(key)
endRead()
numReaders=0;
numWriters=0;
signal()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()

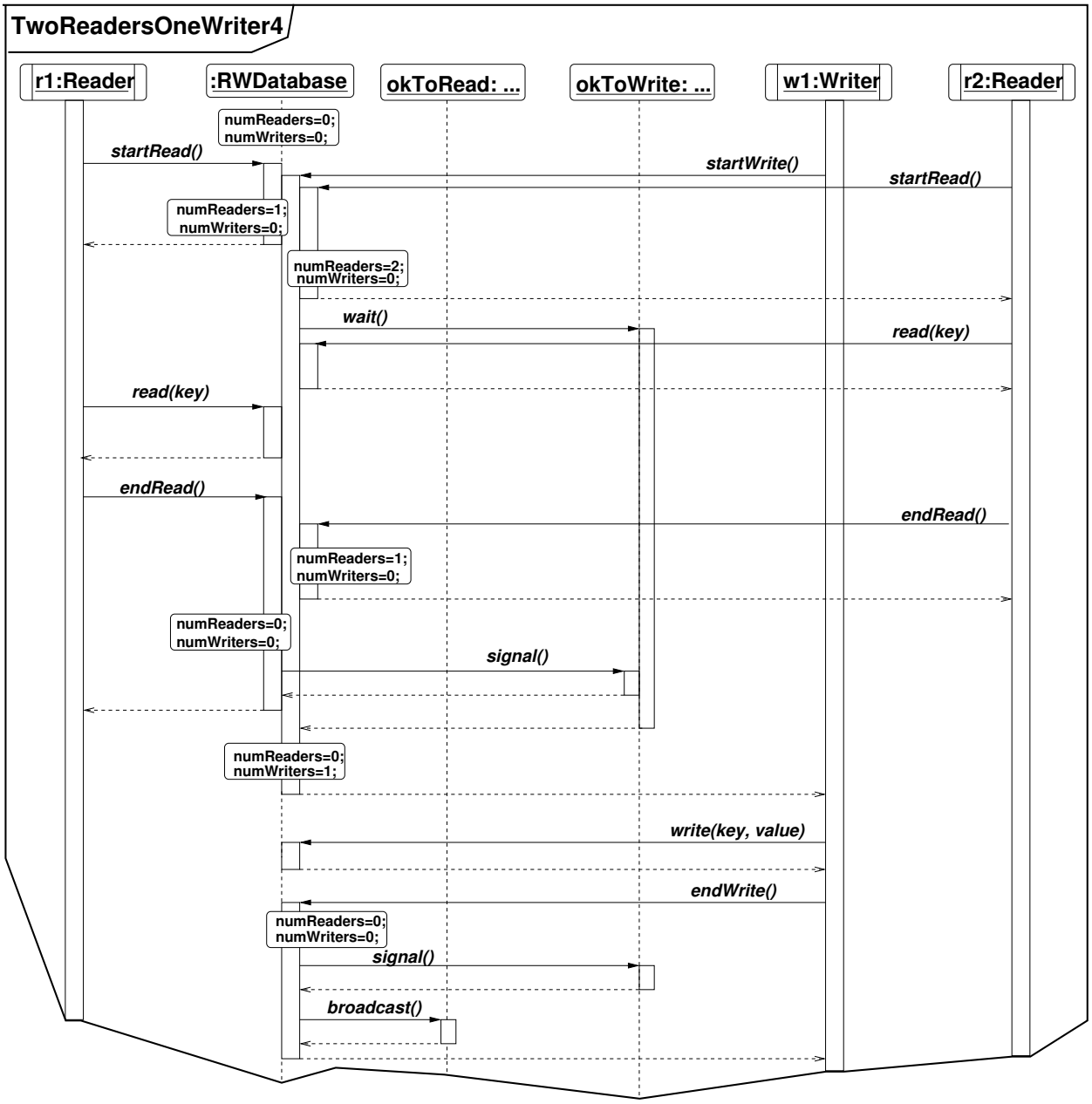

OneReaderOneWriter3
w1:Writer
:RWDatabase
okToRead: ...
okToWrite: ...
r1:Reader
numReaders=0;
numWriters=0;
startWrite()
numReaders=0;
numWriters=1;
startRead()
wait()
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()
numReaders=1;
numWriters=0;
read(key)
endRead()
numReaders=0;
numWriters=0;
signal()

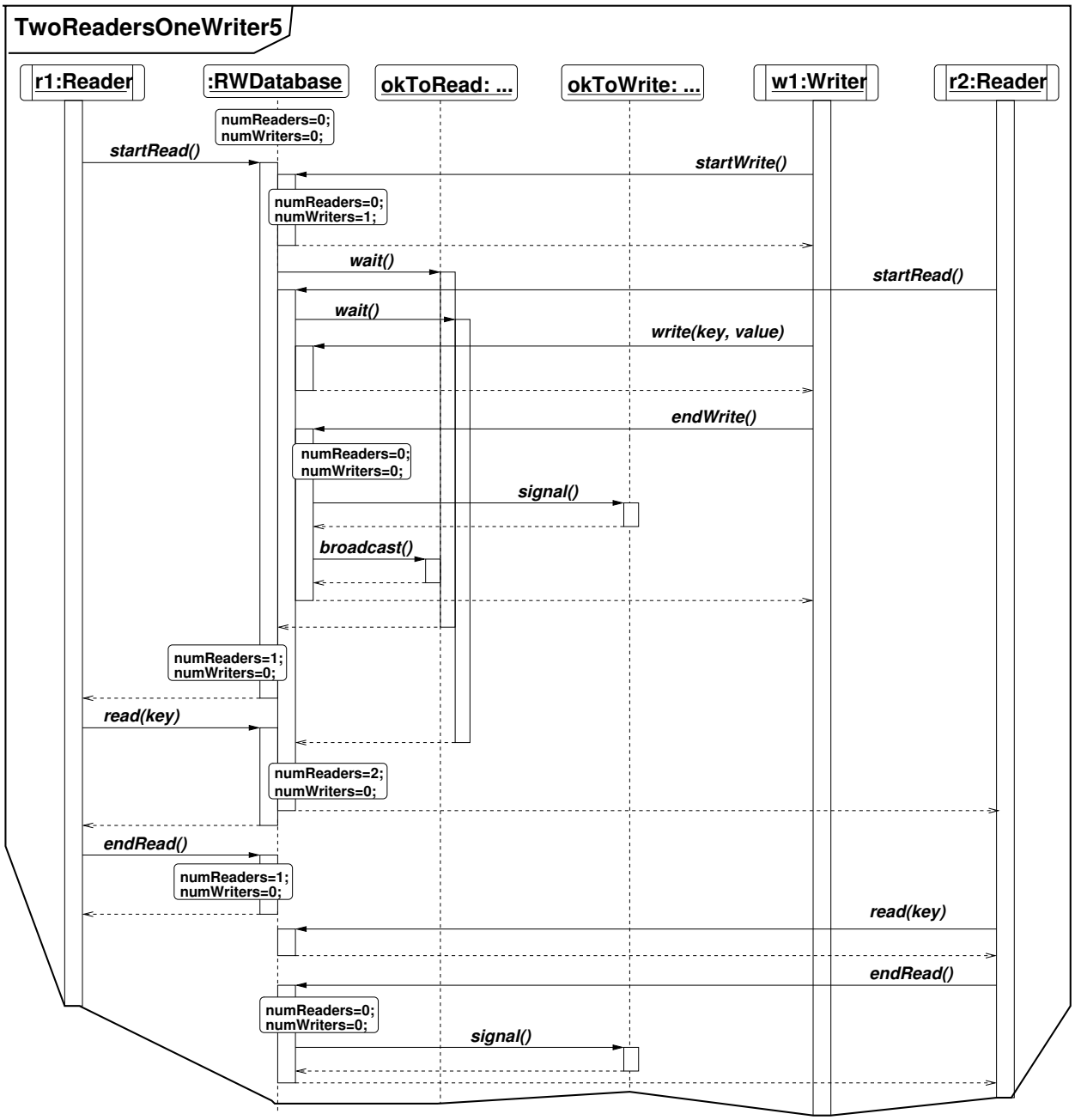

OneReaderOneWriter4
w1:Writer
:RWDatabase
okToRead: ...
okToWrite: ...
r1:Reader
numReaders=0;
numWriters=0;
startWrite()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
startRead()
numReaders=0;
numWriters=0;
signal()
broadcast()
numReaders=1;
numWriters=0;
read(key)
endRead()
numReaders=0;
numWriters=0;
signal()

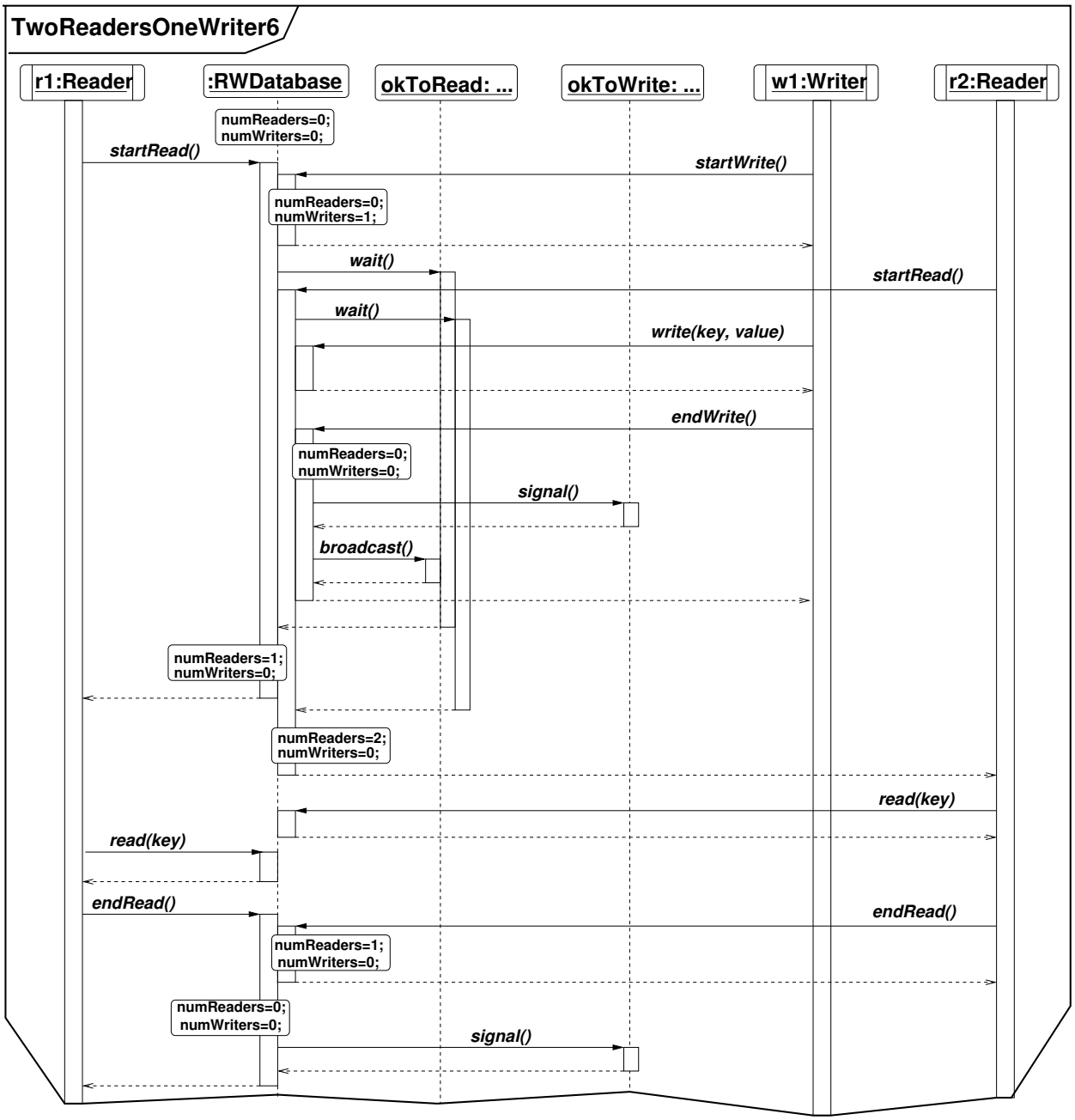

TwoReadersOneWriter1
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
w1:Writer
r2:Reader
numReaders=0;
numWriters=0;
startRead()
numReaders=1;
numWriters=0;
read(key)
startWrite()
wait()
startRead()
endRead()
numReaders=2;
numWriters=0;
numReaders=1;
numWriters=0;
read(key)
endRead()
numReaders=0;
numWriters=0;
signal()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()

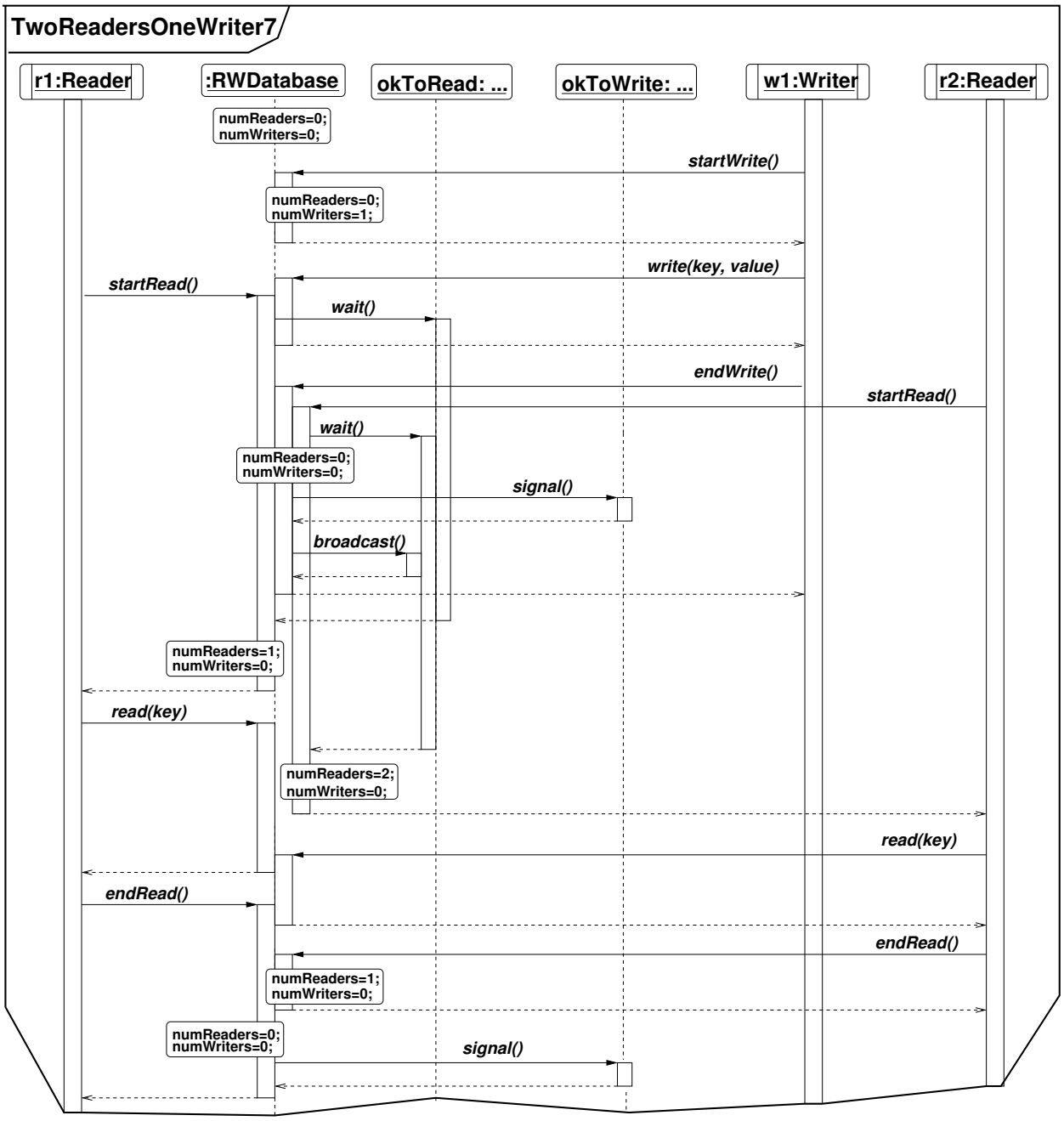

TwoReadersOneWriter2
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
w1:Writer
r2:Reader
numReaders=0;
numWriters=0;
startRead()
numReaders=1;
numWriters=0;
startWrite()
wait()
read(key)
startRead()
numReaders=2;
numWriters=0;
endRead()
read(key)
numReaders=1;
numWriters=0;
endRead()
numReaders=0;
numWriters=0;
signal()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()

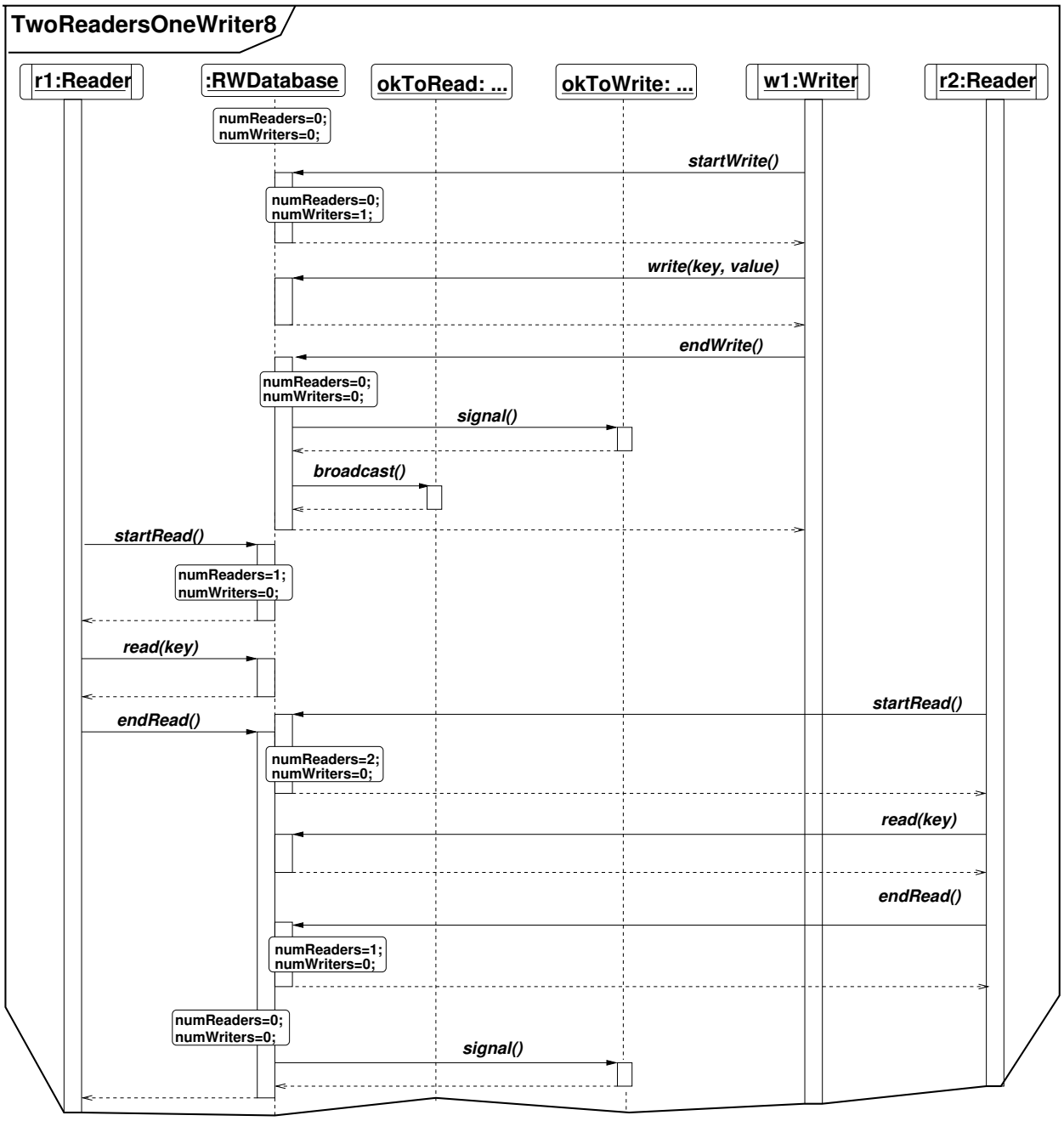

TwoReadersOneWriter3
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
w1:Writer
r2:Reader
numReaders=0;
numWriters=0;
startRead()
numReaders=1;
numWriters=0;
startWrite()
wait()
read(key)
startRead()
numReaders=2;
numWriters=0;
read(key)
endRead()
numReaders=1;
numWriters=0;
endRead()
numReaders=0;
numWriters=0;
signal()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()


TwoReadersOneWriter4
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
w1:Writer
r2:Reader
numReaders=0;
numWriters=0;
startRead()
startWrite()
startRead()
numReaders=1;
numWriters=0;
numReaders=2;
numWriters=0;
wait()
read(key)
read(key)
endRead()
endRead()
numReaders=1;
numWriters=0;
numReaders=0;
numWriters=0;
signal()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()


TwoReadersOneWriter5
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
w1:Writer
r2:Reader
numReaders=0;
numWriters=0;
startRead()
startWrite()
numReaders=0;
numWriters=1;
wait()
startRead()
wait()
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()
numReaders=1;
numWriters=0;
read(key)
numReaders=2;
numWriters=0;
endRead()
numReaders=1;
numWriters=0;
read(key)
endRead()
numReaders=0;
numWriters=0;
signal()


TwoReadersOneWriter6
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
w1:Writer
r2:Reader
numReaders=0;
numWriters=0;
startRead()
startWrite()
numReaders=0;
numWriters=1;
wait()
startRead()
wait()
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()
numReaders=1;
numWriters=0;
numReaders=2;
numWriters=0;
read(key)
read(key)
endRead()
endRead()
numReaders=1;
numWriters=0;
numReaders=0;
numWriters=0;
signal()


TwoReadersOneWriter7
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
w1:Writer
r2:Reader
numReaders=0;
numWriters=0;
startWrite()
numReaders=0;
numWriters=1;
write(key, value)
startRead()
wait()
endWrite()
startRead()
wait()
numReaders=0;
numWriters=0;
signal()
broadcast()
numReaders=1;
numWriters=0;
read(key)
numReaders=2;
numWriters=0;
read(key)
endRead()
endRead()
numReaders=1;
numWriters=0;
numReaders=0;
numWriters=0;
signal()


TwoReadersOneWriter8
r1:Reader
:RWDatabase
okToRead: ...
okToWrite: ...
w1:Writer
r2:Reader
numReaders=0;
numWriters=0;
startWrite()
numReaders=0;
numWriters=1;
write(key, value)
endWrite()
numReaders=0;
numWriters=0;
signal()
broadcast()
startRead()
numReaders=1;
numWriters=0;
read(key)
startRead()
endRead()
numReaders=2;
numWriters=0;
read(key)
endRead()
numReaders=1;
numWriters=0;
numReaders=0;
numWriters=0;
signal()