2010 IEEE Symposium on Visual Languages and Human-Centric Computing

Debugging Concurrent Software: A Study Using Multithreaded Sequence Diagrams*

Scott D. Fleming[1,4], Eileen Kraemer[2], R. E. K. Stirewalt[3,4], Laura K. Dillon[4]

[1]Oregon State University
sdf@eecs.oregonstate.edu

[2]Univ. of Georgia
eileen@cs.uga.edu

[3]LogicBlox, Inc.
kurt.stirewalt
@logicblox.com

[4]Michigan State Univ.
ldillon@cse.msu.edu

Abstract

Concurrent software is notoriously difficult to debug. We investigate the use of UML sequence diagrams to help developers correctly reason about the potential behaviors of buggy concurrent software. We conducted a controlled experiment that compared internal (i.e., "in the head") and external representations for reasoning about multithreaded software. For external representations, participants created multithreaded sequence diagrams. The results of the experiment demonstrate a strong positive effect associated with using external representations. Participants who drew diagrams were significantly more successful at reasoning about the potential behavior of concurrent software. Moreover, participants who produced diagrams with higher levels of detail and with fewer errors tended to achieve greater levels of success. Additionally, this paper contributes an extension to the UML sequence diagram notation for showing behavior of multithreaded software and formal metrics for assessing the complexity of thread interactions.

1. Introduction

Concurrency can provide important performance benefits to software systems; however, it also substantially increases the complexity of software. This complexity makes debugging particularly difficult. Programmers commonly debug programs by replaying a failure repeatedly to understand how it occurs [6] and by tracing data and control dependences backward through the code to find the source of an error [17]. Unfortunately, concurrency renders these techniques ineffective: failures may be difficult to reproduce, and multiple data and control flows make dependences difficult to understand. To make matters worse, developers lack specialized debugging tools in practice [8], and the literature is largely silent on what techniques are effective for debugging concurrent software. Our work seeks to understand the strategies that successful programmers use when debugging concurrent software and to develop tools and techniques to support those strategies.

A recent study [4] found that successful programmers modeled the potential behaviors of concurrent programs to explain how errors occurred. Such models described *thread interactions*, scenarios of multithreaded program behavior, such that the instructions executed by the threads form a sequence and are interleaved in accordance with some thread schedule.

Although modeling was associated with success on debugging tasks, the study also found limitations. Some programmers who engaged in modeling were unable to produce a model that explained the failure, thus limiting their success on task. Model complexity may have been an important factor in such cases. Programmers in the study created models predominantly *internally*—that is, "in their heads." Modeling in this manner may have strained the programmers' cognitive resources, such as *working memory* [1], and thus hampered their ability to reason about how the execution of program instructions affects the system's state throughout the interaction.

Research in distributed cognition and external cognition offers insight into how external representations can help with such information processing tasks. When working on such tasks, representing relevant information *externally*—that is, using the external environment—can provide cognitive benefits, such as reducing cognitive load [15]. For example, information visualization can help a human offload information processing to the visual representation [5]. However, the form that an external representation takes can influence how helpful the representation is in problem solving [21]. Even different *isomorphic representations*—that is, representations that encode the same information—can cause very different cognitive behaviors.

*This material is based in part upon work supported by the National Science Foundation under Grant Numbers CCF-0702667 and IIS-0308063. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

978-0-7695-4206-5/10 $26.00 © 2010 IEEE
DOI 10.1109/VLHCC.2010.14

IEEE computer society

Thus, when proposing a new diagram notation, we must answer questions about the benefits of diagram construction for this task, and must consider the effect of time on task. Is this notation useful for this task? Might the time required to construct the diagram be better spent in just thinking about the problem?

In this paper, we investigate how creating external representations in the form of UML sequence diagrams [14] influences a programmer's ability to reason about the potential behaviors of a concurrent program. In particular, we seek to address one primary and two subsidiary research questions:

RQ1: To what extent does creating sequence diagrams improve a programmer's ability to reason about the potential behaviors of a buggy multithreaded program?

RQ2: Does a relationship exist between the complexity of thread interactions and the effect of externalizing on reasoning ability?

RQ3: Does a relationship exist between the quality of a diagram and the effect of externalizing on reasoning ability?

We focus on the UML sequence diagram notation because it is well known and widely used.

This paper makes the following contributions: (1) a controlled experiment investigating the use of extended UML sequence diagrams for reasoning about concurrent software; (2) a multithreaded extension to the UML sequence diagram notation; and (3) formal metrics for assessing the complexity of thread interactions.

The remainder of the paper is organized as follows. Section 2 provides background on external representations of thread interactions and presents our multithreaded sequence diagrams. Section 3 presents formal definitions for our complexity metrics. Section 4 describes our experimental method. Section 5 presents the results of our experiment. Section 6 discusses threats to validity. Finally, Section 7 closes the paper with discussion and conclusions.

2. Representing Thread Interactions

Although UML sequence diagrams are widely used for representing object interactions, we found the notation inadequate for expressing some intricacies of thread interactions. In this section, we describe the UML sequence diagram notation and our extension to address the inadequacies. Finally, we contrast our notation with other relevant notations and visualizations from the literature.

UML sequence diagrams. Sequence diagrams are a popular notation in the Unified Modeling Language [14] for representing scenarios of execution by displaying the sequence of messages exchanged between objects. Fig. 1 depicts a sequence diagram. The vertical dimension represents time, which proceeds down the page. Each object is drawn in a vertical column that contains a head symbol and a vertical lifeline. Fig. 1 depicts an interaction between two objects: obj1 and obj2. The notation defines two kinds of objects: *active objects*, such as obj1, which have an associated thread of control, and *passive objects*, such as obj2, which do not. An active object is distinguished by a double line on each side of its head symbol. An object's lifeline may be adorned with *object states* to denote changes in the state of the object. An object state appears as a rounded rectangle that contains a description of the new state (typically in the form of a predicate). For example, the obj2 attribute var1 initially has the value 0.

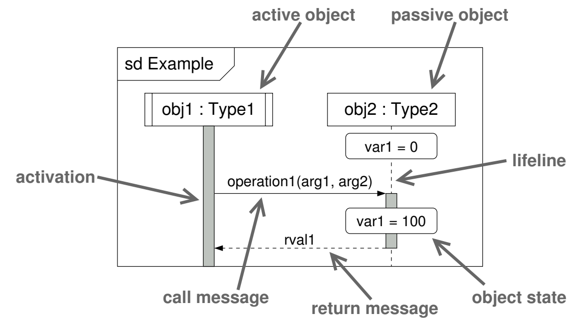

Figure 1. Example of a UML sequence diagram. The labels around the perimeter are for exposition and are not part of the diagram.

A message is shown as an arrow from the lifeline of one object to that of another. An arrow with a solid line depicts a call to an operation and may be adorned with the operation name and arguments. For example, obj1 calls the operation operation1 on obj2, passing arg1 and arg2 as arguments. *Activations* represent the execution of methods and are depicted as bars that overlap the lifelines. For example, during the activation of operation1, var1 becomes 100. An active object, such as obj1, executes continuously throughout its lifetime, so an activation bar always covers its lifeline. A message arrow with a dashed line depicts a return and may be adorned with a return value. In the example, rval1 is returned upon the completion of operation1.

The sequence diagram notation lacks convenient features for expressing properties of multithreaded systems, such as when each thread runs and blocks, and when context switches occur. To address the lack of support for multithreading in standard UML, we designed a multithreaded extension to the sequence diagram notation.

Multithreaded extension to UML sequence diagrams. Our extension has several new notations to represent thread state. Fig. 2 depicts an example that uses our extension. A hatched activation bar denotes that the thread associated with the activation is in the blocked state. A non-hatched activation bar denotes that the thread is in the ready or running state. A thread name (L, H1, or H2) along the left side of the diagram denotes the thread that is running. A hori-

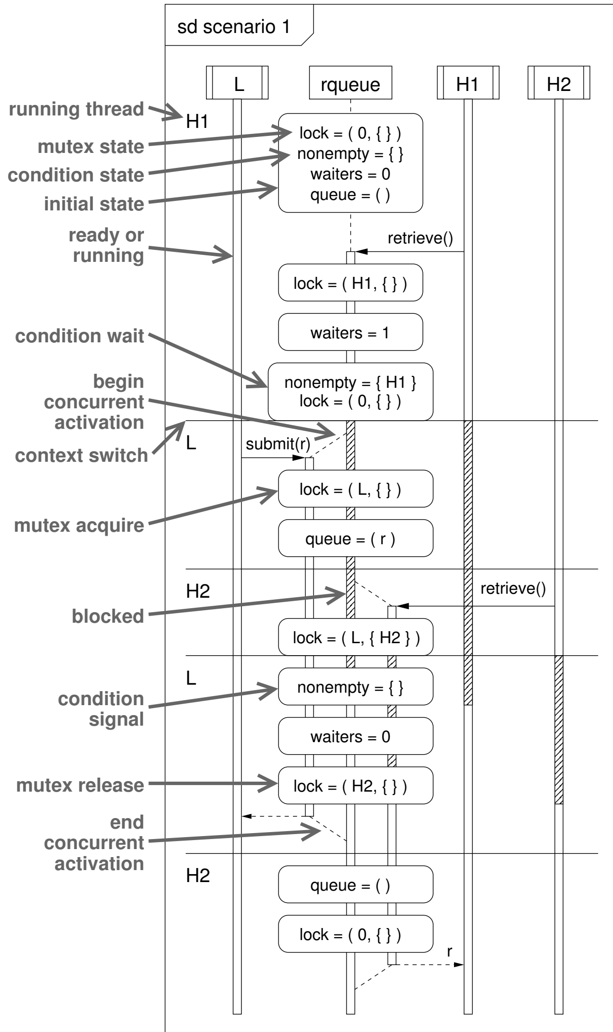

Figure 2. Multithreaded sequence diagram.

zontal line that crosscuts the entire diagram denotes a context switch, a point when the running thread changes. A branch of an object's lifeline denotes a concurrent activation (e.g., see rqueue's lifeline following the first context switch in Fig. 2). When a concurrent activation ends, the branch merges back into the lifeline.

We leverage object states to denote the effects of operations on mutexes and condition variables. The pair (h,W) denotes the state of a mutex where h is the holder of the mutex (0 indicates that the mutex is not held), and W is the set of threads waiting on the mutex. For example, in Fig. 2 the first object state after L calls submit (i.e., lock = (L,$\{\}$)) indicates that L has acquired the mutex lock and no threads are waiting for the lock. The object state immediately before L returns from submit indicates that L has released the lock and H2 has acquired it. The state of a condition variable is denoted by a set of waiting threads. For example, the object state immediately before the first context switch indicates that H1 begins waiting on the condition variable nonempty. This object state also depicts the release of lock by H1.

Other visualizations. Approaches to the visualization of thread interactions can be classified as either static or dynamic. Static visualizations involve fixed images, whereas dynamic visualizations involve animations. Our current work focuses on static visualizations.

Schader and Korthaus [16] described features of UML that support the representation of concurrency, with an emphasis on Java thread behavior. They describe how the style of arrow used in sequence diagrams can convey information about the type of communication between objects, the use of staggered activations to represent a series of iterative calls, and the use of labeling, such as "$*||[i := 1..n]$," to denote n methods executing in parallel. They state that conditional branches [14] may also denote concurrency if the guard conditions are not mutually exclusive. In contrast to our extension, they offer no conventions for explicitly denoting thread states, such as blocking, or the state of synchronization mechanisms, such as mutexes.

Others have extended sequence diagrams to better support concurrent software. Mehner and Wagner [11, 12] added shading conventions on activations to indicate when, and within which activation, threads are ready or running. In contrast to our extension, their extension omits some thread information, such as which thread is running and when context switches occur. Their extension is geared toward Java and includes calls to a distinguished `synchronize` operation. It depicts when threads block on entry into methods, which makes deadlocks easier to recognize. However, unlike our extension, theirs abstracts away details, such as when locks are released during condition synchronization.

Xie et al. [19, 20] developed a concurrent extension to sequence diagrams that focuses on the depiction of monitor objects. In their extension, the color (green, yellow, red) of activation bars indicates thread status (running, ready, suspended); object states indicate monitor status (locked, unlocked); and comments indicate the state of variables within a monitor. Thread interleavings and context switches can be easily viewed by following the trace of green activation bars. However, their use of color makes the diagram inconvenient to draw by hand with a pen or pencil, and precludes the use of colors to distinguish the activations of different threads—a feature that our extension supports. Furthermore, their extension does not explicitly represent mutexes or condition variables, nor the threads waiting on locks and condition variables, as we do in our extension.

Newman et al. [13] proposed two diagramming notations to statically visualize concurrency-related design decisions. Their *regional state hierarchy diagram* extends the UML class diagram to depict the structure of lock-state associations. Their *method concurrency diagram* extends a call graph to provide details on calls to and data protected by

mutexes and condition variables. Although the Newman diagrams do not model thread interactions explicitly, they provide supporting information that could be useful in conjunction with, for instance, a sequence diagram.

The literature contains numerous dynamic visualizations of thread interactions. Traditional parallel debuggers (e.g., [10]) provide a rudimentary visualization by displaying a debugger window for each thread. Other visualization tools show the status of various properties as a multithreaded program executes. Leroux et al. [7] developed a tool that dynamically elaborates a standard UML sequence diagram, and as the diagram grows, the tool also displays the current state of each thread. Although static visualizations are our current focus, we plan to investigate dynamic visualizations in future work.

3. Assessing Thread-Interaction Complexity

RQ2 explores the relationship between the complexity of thread interactions and the effect of externalizing; however, the literature offers no metrics for assessing interaction complexity. We have identified three properties that we suspect contribute to the complexity of a thread interaction: (1) the number of threads involved in the interaction, (2) the number of times threads block or unblock, and (3) the number of context switches. In this section, we define rigorous metrics for these properties in terms of labeled transition system (LTS) models of multithreaded programs.

Modeling multithreaded programs as LTSs. LTSs have been widely used to model concurrent programs [9]. In such models, states encode limited history and actions represent atomic program instructions. For example, Fig. 3 depicts an LTS model, which we refer to as *2ThreadMutex*, of a multithreaded program that comprises two threads, t_1 and t_2, and one mutex. Starting from the initial state, s_0, each thread infinitely acquires and releases a shared lock. Given an LTS model of a multithreaded program, a thread interaction can be modeled as a *trace*, which comprises a sequence of states and actions produced by executing the model. For example, the following trace, *2TMTrace*, represents one possible execution of the *2ThreadMutex* model:

$$s_0 \xrightarrow{t_1.acquire} s_1 \xrightarrow{t_2.tryAcquireBlock} s_3$$
$$\xrightarrow{t_1.releaseUnblockT2} s_2 \xrightarrow{t_2.release} s_0$$

Methods for producing a *faithful* LTS model from a multithreaded program are well-documented (e.g., [9]). To define our metrics, we require a faithful LTS model with two additional properties. First, each transition in the model must represent execution of one and only one thread. Second, blocking and unblocking actions by threads must be modeled explicitly. The *2ThreadMutex* model possesses these properties. Each action models the execution of either thread t_1 or t_2. Also, the actions $t_1.tryAcquireBlock$ and

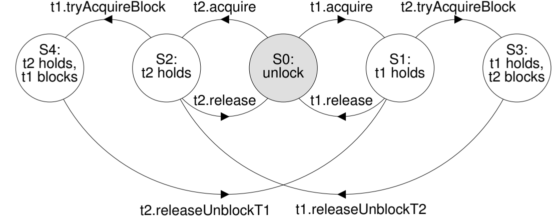

Figure 3. LTS model of a program in which two threads acquire and release a shared lock.

$t_2.releaseUnblockT2$ explicitly model t_1 entering and exiting the blocking state, respectively. Thread t_2 executes similar blocking/unblocking actions.

Formal definitions. Let *States* be the universal set of states, *Threads* be the universal set of threads, and *Actions* be the universal set of actions. A finite multithreaded LTS P is a 6-tuple $\langle S, q, T, A, \Theta, \Delta \rangle$ where $S \subseteq States$ is a finite set of states; $q \in S$ indicates the initial state of P; $T \subseteq Threads$ is a finite set of threads; $A \subseteq Actions$ is a finite set of actions; $\Theta \subseteq A \times T$ is a total function that maps each action to the thread that executes the action; and $\Delta \subseteq S \times A \times S$ is a transition relation that maps a source state and action pair to a target state. For example, the following 6-tuple represents the *2ThreadMutex* LTS:

$$S = \{ s_0, s_1, s_2, s_3, s_4, s_5 \} \qquad q = s_0 \qquad T = \{ t_1, t_2 \}$$
$$A = \{ t_1.acquire,\ t_2.acquire,\ \ldots\}$$
$$\Theta = \{ (t_1.acquire, t_1),\ (t_1.tryAcquireBlock, t_1),\ \ldots \}$$
$$\Delta = \{ (s_0, t_1.acquire, s_1),\ (s_0, t_2.acquire, s_2),\ \ldots \}$$

We define a trace R of an LTS P to be a tuple $\langle R_S, R_A \rangle$ where R_S is a sequence over S; R_A is a sequence over A such that $|R_A| = |R_S| - 1$; $R_S(1) = q$; and $\forall i \in \{1 \ldots |R_A|\}, (R_S(i), R_A(i), R_S(i+1)) \in \Delta$. For instance, the following represents *2TMTrace*:

$$R_S = \langle\ s_0, s_1, s_3, s_2, s_0\ \rangle$$
$$R_A = \langle\ t_1.acquire, t_2.tryAcquireBlock,$$
$$t_1.releaseUnblockT2, t_2.release\ \rangle$$

Given a trace R of P and a set of interesting actions $B \subseteq A$, we define the following sets: (1) The set of threads involved in R, $threadSet(R) = \bigcup_i \Theta(R_A(i))$; (2) the set of indices of R at which an action in B occurs, $occurs_B(R) = \{i | R_A(i) \in B\}$; and (3) the set of indices of R at which a context switch occurs $threadSwitch(R) = \{i | \Theta(R_A(i)) \neq \Theta(R_A(i-1))\}$.

Using these sets, we define three metrics for a given trace R of an LTS P: (1) The number of threads involved in R is $|threadSet(R)|$; (2) the number of times threads block or unblock in R is $|occurs_{Block}(R) \cup occurs_{Unblock}(R)|$, where $Block \subseteq A$ and $Unblock \subseteq A$ denote the sets of block actions and unblock actions, respectively; and (3) the number of

context switches that occur in R is $|threadSwitch(R)|$. For example, in *2TMTrace* the number of threads is 2, the number of blocks and unblocks is 2, and the number of context switches is 3. Given a multithreaded LTS model, these metrics are trivial to compute for any trace R.

4. Experimental Method

The primary goal of our experiment was to investigate the extent to which creating UML sequence diagrams improves programmers' ability to reason about concurrent program executions (RQ1). In this section, we provide the details of our experimental method.

Participants. The participants comprised 44 undergraduate CS students enrolled in a software design course at Michigan State University. All participants had previously completed at least one course that emphasized C++ programming. The software design course that participants were taking covered principles of good design, design patterns, and object-oriented programming in C++. We did not collect data on concurrent-programming experience, but measured concurrent-programming ability instead with a prequestionnaire (described next).

Materials. The study materials comprised a prequestionnaire and an experiment questionnaire (full versions in [3]).

The prequestionnaire contained 8 questions that measured ability to reason about the potential behaviors of a multithreaded program. To answer the questions correctly required an understanding of the multithreaded programming model.

The experiment questionnaire, like the prequestionnaire, measured the ability to reason about the potential behaviors of a multithreaded program, but it involved more complex thread interactions. The questions referred to a small (56 SLOC) multithreaded server program, which we seeded with a defect. The program simulates an e-business server that accepts and processes requests from remote clients. It comprises multiple threads, each of which plays one of two distinct roles. A single *listener* thread monitors the network, listening for client requests and placing them on a request queue as they arrive. Two *handler* threads take requests from the request queue and simulate the processing of the requests. A race condition allows a handler to erroneously invoke the pull operation on an empty request queue.

The questionnaire presented four different scenarios, each followed by a question asking whether the scenario is consistent with the code and whether the program would enter an error state during the scenario. Fig. 4 reproduces one of the scenarios from the questionnaire. (The multithreaded sequence diagram in Fig. 2 represents this scenario.) The scenario elides many details of the interaction; however, it contains enough detail to be unambiguous.

Scenario: Assume there is a listener thread, L, and two handler threads, H_1 and H_2, and that

- `queue` is empty,
- `waiters` is zero, and
- all the threads are at the beginning of their respective control loops.

Consider the scenario where:

(1) H_1 calls `retrieve` and blocks inside the operation.
(2) L calls `submit` (with argument r) and is preempted at line 17.
(3) H_2 calls `retrieve` and blocks inside the operation.
(4) L returns from `submit` and is preempted at the top of its control loop. In the process, H_1 transitions to the ready state.
(5) H_2 returns from `retrieve` and is preempted at the top of its control loop.

Question: Is the scenario consistent with the code? If so, does the scenario result in the program entering an error state? (Select one of the following.)

(a) **Consistent & No Error:** The scenario *is consistent* with the code and *does not* result in the program entering an error state.
(b) **Consistent & Error:** The scenario *is consistent* with the code and *does* result in the program entering an error state.
(c) **Inconsistent:** The scenario *is not consistent* with the code.

Figure 4. A scenario from the experiment questionnaire.

Experiment design. Our experiment had one independent variable—the use of external representations; two treatments—the exclusive use of internal representations (*internal*) and the use of external representations in the form of sequence diagrams (*external*). These treatments were inspired by a prior study [4] which found that, despite a having pencil and paper readily at hand, programmers seldom represented thread interactions externally, instead preferring to reason about such interactions "in their heads." Our experiment also had one dependent variable—the ability to reason correctly about thread interactions (C). We measured C as the proportion of correct answers on the experiment questionnaire (i.e., as a score of 0, 0.25, 0.5, 0.75, or 1.0). Our null hypothesis (H_0) was that externalizing has no effect on ability to reason ($C(external) = C(internal)$). Our alternative hypothesis (H_a) was that externalizing yields better scores on the questionnaire than using only internal representations ($C(external) > C(internal)$). We did not anticipate the result $C(external) < C(internal)$ because the external group also had their internal facilities. If we observed such a result, we would assume it was due to random chance or a defect in the experiment. To test our hypotheses, we used a *between-subjects* design (i.e., each participant received one of the treatments).

Procedure. Our study comprised three 80-minute sessions spread over two weeks. During the first two sessions, participants received a two-part lecture on multithreaded

Table 1. Questionnaire results.

Question	External ($N = 22$)		Internal ($N = 22$)		Diff in M
	M	SD	M	SD	
1	0.59	0.50	0.32	0.48	0.27
2	0.64	0.49	0.14	0.35	0.50
3	0.55	0.51	0.45	0.51	0.09
4	0.41	0.50	0.32	0.48	0.09
Overall	0.55	0.31	0.31	0.27	0.24

programming in C++. Part of the lecture covered our multithreaded sequence-diagram extension. During the last 30 minutes of the second session, participants filled out the pre-questionnaire. We collected and graded 50 completed pre-questionnaires. Throughout the study, we anonymized all materials, replacing participant names with ID numbers.

Between the second and third sessions, we used the pre-questionnaire scores to partition the participants into two balanced treatment groups: the *external group* and the *internal group*. Six of the original 50 participants dropped out of the study, but the attrition did not unbalance the groups: the final groups were of equal size (22) and had equal pre-questionnaire means (0.494). The standard deviation varied only slightly between the groups (external = 0.260; internal = 0.236).

For the final session, the treatment groups met in separate rooms and filled out the experiment questionnaire. We asked the external group to draw a sequence diagram of each scenario before answering questions about that scenario. In contrast, we asked the internal group to answer the questions "in their heads"—that is, without drawing pictures or writing notes. We collected 44 completed questionnaires along with all drawings and notes made by participants. An inspection of the internal group's materials revealed no evidence of externalizing.

5. Results

In this section, we present results of our experiment. First, we assess the extent to which externalizing models of thread interactions with multithreaded sequence diagrams improves ability to reason correctly about the potential behavior of a buggy multithreaded program (RQ1). Next, we analyze the complexity of the interactions to see if there is a relationship with the differences in scores of the two treatment groups (RQ2). Last, we analyze the quality of diagrams produced by the external group to see if there is a relationship with performance on the questionnaire (RQ3).

Effects of externalizing (RQ1). Table 1 and Fig. 5 depict the descriptive statistics for the experiment questionnaire. Table 1 summarizes the results for the questions individually as well as in the aggregate.

To test our hypothesis H_a, we used a two-sample t test

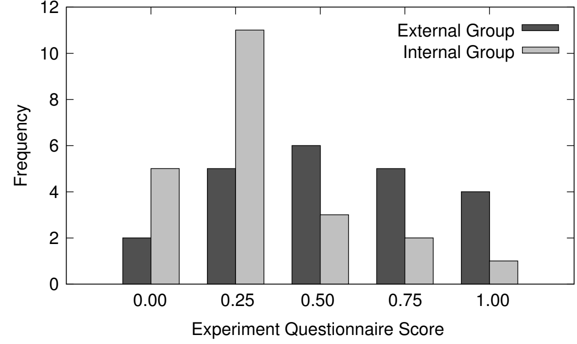

Figure 5. Frequency distribution of overall scores.

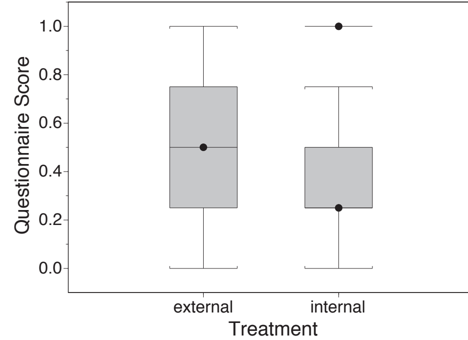

Figure 6. Box plot of scores. For the internal group, the median equaled the lower-quartile boundary. The internal group also contained one outlier with a score of 1.0.

($\alpha = 0.05$) that compared overall questionnaire scores in external and internal conditions. We used a one-tailed t test because (1) we predicted that the external group would have a higher mean prior to executing the study, and (2) if the internal group produced a higher score, we would attribute the difference to random chance. The external group scored significantly higher than the internal group ($t(40.89) = 2.71$, $p = 0.005$).[1] Fig. 6 depicts the difference as a box plot. Based on these results, we accepted our alternative hypothesis H_a that creating multithreaded sequence diagrams improves ability to reason about thread interactions.

Interaction complexity (RQ2). Looking at the individual questions, the external group scored higher than the internal group on every question; however, the effect of externalizing appeared to be stronger on some questions (1 and 2) than others (3 and 4). We conducted a post-hoc analysis to check for a relationship between this effect and interaction complexity. In particular, we applied the three metrics from Section 3 to the four scenarios of interaction from the questionnaire. We first modeled the program from the questionnaire as an LTS in the FSP language [9]. The resulting model had 1097 states and 2416 transitions (details in [3]). Next, we computed a trace that represented each

[1] As a validity check, we also computed the (nonparametric) Wilcoxon rank-sum test ($Z = 2.5741$, $p = 0.005$), which confirmed the t-test results.

Table 2. Complexity measurements over the questionnaire scenarios. (All involved 3 threads.)

Scenario	Transitions	Context Switches	(Un)Blocks
1	22	4	4
2	11	3	4
3	15	2	1
4	13	2	1

interaction using the LTSA tool.[2]. For each trace, we computed the number of threads involved, the number of context switches, and the number of block/unblock actions. We also computed the total number of transitions, which is relevant to complexity but is not specific to concurrency.

Our sample was too small for statistical analysis; however, Table 2 shows that the number of context switches and block/unblock actions were noticeably higher for questions 1 and 2 than for questions 3 and 4. It is possible that these differences could account for the differences in the effect of externalizing because a few context switches or block-/unblock actions can dramatically increase the amount of program state that a person must keep track of to reason correctly about an interaction. We could not observe a relationship between the number of threads and the effect of externalizing because the number did not vary (each question involved 3). Although the number of transitions varied between scenarios, there was no apparent relationship between the metric and the effect of externalizing. These results suggest a relationship between interaction complexity and the effect of externalizing. In particular, the effect of externalizing appears to increase with the number of context switches and block/unblock actions.

Diagram quality (RQ3). To address RQ3, we checked for a correlation between diagram quality and questionnaire score. One researcher evaluated the diagrams using a detailed rubric (given in [3]), which emphasized not only the correctness of a diagram, but also the level of detail. For example, diagrams that did not explicitly represent state changes to the mutex objects received a lower score than those that did. The rater assigned 1 point for each feature present and correct in a diagram (9 or 10 max depending on the question). He did not require participants to use our sequence-diagram extension, and did not deduct points for incorrect or non-standard syntax. Furthermore, he did not know the participants' questionnaire scores or who each participant was; only numeric IDs appeared on the diagrams. Each participant received an overall diagram quality score that was the mean of the percentage scores on the individual diagrams.

Our analysis revealed a significant correlation between diagram quality and questionnaire score (Pearson: $r(20) = 0.692$, $p = 0.0004$). In particular, as diagram quality increased, the ability to reason about models also increased. Fig. 7 depicts this relationship graphically.

[2] By design, one such trace existed for each question

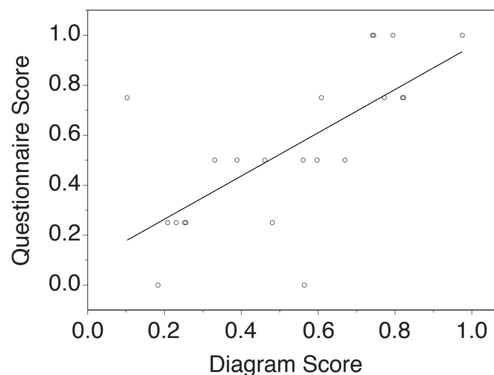

Figure 7. Plot of the relationship between diagram quality and questionnaire score.

6. Threats to Validity

Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variable. One threat to internal validity was that participants may have improved their mastery of concurrent programming between the time that they completed the prequestionnaire and the experiment questionnaire—in fact, simply completing the prequestionnaire may have caused them to learn. We minimized this threat by keeping the time between the prequestionnaire and experiment questionnaire short (i.e., less than a week). Another threat to internal validity could have arisen if one group or the other resented either the lack of diagrams or the additional work of creating them, and underperformed. We addressed this threat by administering the experiment questionnaire to the groups simultaneously in separate rooms.

Construct validity is the degree to which the independent and dependent variables accurately measure the concepts they purport to measure. A threat to construct validity arose because the theoretical construct of reasoning ability is difficult to operationalize. Our questions measured aspects of reasoning ability that we found interesting; however, there may be aspects we did not measure.

External validity is the degree to which the results of the research can be generalized to the population under study and other research settings. Threats to external validity arose because our participants were students and the programs and scenarios were kept small to meet the time constraints of our study. In future work, we will address the threats to external validity by conducting additional studies that involve a wider range of subjects and contexts. Obtaining professional participation is difficult and costly. To address this, we will use different types of studies (e.g., case studies) that allow for small samples and large-scale artifacts in exchange for reduced control of variables. This ap-

proach is based on the idea that the weaknesses of one empirical study can be addressed by the strengths of another study [18]. If subsequent studies confirm our results, then we can have confidence that our findings generalize.

7. Discussion and Conclusions

In conclusion, our experimental results demonstrated that creating external representations in the form of multithreaded sequence diagrams substantially improved programmers' ability to reason correctly about a buggy multithreaded program (RQ1). Participants who created sequence diagrams scored nearly a full standard deviation higher than those who used only internal representations. The difference was statistically significant, and we were able to reject the null hypothesis ($p = 0.005$).

Our results also suggest a relationship between interaction complexity and the benefit of creating sequence diagrams (RQ2). We observed that the effect of externalizing was noticeably greater for the interactions with greater numbers of context switches and block/unblock operations. This observation supports the idea that creating sequence diagrams alleviates the cognitive load associated with reasoning about complex thread interactions. It is an open question whether there is a threshold of complexity beyond which programmers tend to encounter cognitive overload.

Lastly, our results show a strong relationship between diagram quality and reasoning ability (RQ3). Specifically, we found a significant positive correlation between diagram quality and questionnaire scores ($p = 0.0004$). This finding suggests that one way to improve reasoning ability would be to encourage better diagram quality.

The findings of our study suggest that a tool for modeling interactions could aid debugging. Such a tool could provide several important features to help users create highly-detailed and correct sequence diagrams. It could track the synchronization state of the threads and objects to both aid comprehension and detect errors in the diagram. It could enable the user to specify synchronization actions (e.g., acquiring a mutex), and automatically adjust the synchronization state accordingly. By statically analyzing the source code, the tool could automatically check for consistency with the model. We believe this tool could benefit not only novices learning concurrent programming, a subject known to be difficult [2], but also expert software developers.

Our study points to several promising directions for future work: experimentally comparing our sequence diagram notation with other notations, developing a modeling tool to support debugging, empirically validating and refining our interaction-complexity metrics, and generalizing our findings by studying professional software engineers working on large-scale software. Ultimately we aim to design a suite of highly-usable debugging tools for real-world developers of concurrent software.

References

[1] A. D. Baddeley. *Human Memory: Theory and Practice*. Erlbaum, 1990.

[2] Y. Ben-David Kolikant. Learning concurrency: Evolution of students' understanding of synchronization. *Int. J. Hum.-Comput. Stud.*, 60(2):243–268, 2004.

[3] S. D. Fleming. *Successful Strategies for Debugging Concurrent Software: An Empirical Investigation*. PhD thesis, Michigan State Univ., East Lansing, MI, 2009.

[4] 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. ICSE*, pages 759–768, 2008.

[5] J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Sci.*, 11:65–100, 1987.

[6] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36:471–482, 1987.

[7] H. Leroux, A. Réquilé-Romanczuk, and C. Mingins. JACOT: A tool to dynamically visualise the execution of concurrent Java programs. In *Proc. PPPJ*, 2003.

[8] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS*, pages 329–339, 2008.

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

[10] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[11] K. Mehner. JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In *Software Visualization*, pages 163–175. 2002.

[12] K. Mehner and A. Wagner. Visualizing the synchronization of Java-Threads with UML. In *Proc. VL*, pages 199–206, 2000.

[13] E. Newman, A. Greenhouse, and W. L. Scherlis. Annotation-based diagrams for shared-data concurrency. In *Proc. Workshop Concurrency Issues in UML*, 2001.

[14] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison, 2004.

[15] M. Scaife and Y. Rogers. External cognition: How do graphical representations work? *Int. J. Human-Computer Studies*, 45(2):185–213, 1996.

[16] M. Schader and A. Korthaus. Modeling Java Threads in UML. In *The Unified Modeling Language: Technical Aspects and Applications*, pages 122–143. Physica, 1998.

[17] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[18] M. Wood, J. Daly, J. Miller, and M. Roper. Multi-method research: An empirical investigation of object-oriented technology. *J. Syst. Softw.*, 48(1):13–26, 1999.

[19] 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. ICSE*, pages 727–731, 2007.

[20] S. Xie, E. Kraemer, and R. E. K. Stirewalt. Empirical evaluation of a UML sequence diagram with adornments to support understanding of thread interactions. In *Proc. ICPC*, pages 123–134, 2007.

[21] J. Zhang and D. A. Norman. Representations in distributed cognitive tasks. *Cognitive Sci.*, 18:87–122, 1994.