Refining Existing Theories of Program Comprehension During Maintenance for Concurrent Software*

Scott D. Fleming[†], Eileen Kraemer[†‡], R. E. K. Stirewalt[†], Laura K. Dillon[†], and Shaohua Xie[‡]

[†]Dept. of Computer Science and Engineering
Michigan State University
East Lansing, Michigan, USA 48824
{sdf,stire,ldillon}@cse.msu.edu

[‡]Department of Computer Science
University of Georgia
Athens, Georgia, USA 30602-7404
{eileen,shaohua}@cs.uga.edu

Abstract

While the sources of complexity in the initial design and verification of multi-threaded software systems are well-documented, less is known of the issues specific to the maintenance of these systems. The literature contains a number of observational studies of programmers performing maintenance, conducted in the context of sequential software and designed to investigate the factors and behaviors that lead to success. To help fill the gap in knowledge in the area of concurrent software maintenance, we conducted a study that refines the findings of two prior studies, those of Littman et al. *and of Vessey, to address issues and obstacles that arise in the understanding of concurrent software. We validated these refinements by observing programmers performing corrective maintenance on a small but complex multi-threaded server program.*

1. Introduction

The design complexities inherent in multi-threaded software systems are well-documented (e.g., [2, 11]). However, much less is known of issues specific to the maintenance of these systems. Previously, we performed a think-aloud study [16] of programmers engaged in the corrective maintenance of a concurrent program [7]. This study strove to understand what behaviors correlate with success on task. In addition to the think-aloud data we collected and used for this prior study, the participants took a posttest to assess the extent to which they understood the program. However, at the time of publication [7], we had yet to analyze these data. This paper reports the results of the posttest analysis and attempts to reconcile these results with the predictions of two well-known studies [10, 17].

Our previous study collected data to uncover the activities programmers perform when diagnosing and correcting synchronization-related faults in multi-threaded programs [7]. The raw data from this previous study comprise video and transcripts collected using the *think-aloud method* [6, 16]. We observed students in a graduate course on formal methods engaged in the corrective maintenance of a faulty system with a realistically complex multi-threaded architecture. The *think-aloud protocols* collected during these observations reveal each participant's "inner dialogue" while he is working on the problem. In addition, each participant completed a posttest, designed to assess various aspects of program comprehension.

This paper attempts to reconcile the phenomena unique to concurrent software with concepts and predictions from two comprehension theories—[10] and [17]—that were originally developed in the context of sequential software. The phenomena unique to concurrency include the potential for data races, subtle control and data flows that arise from thread synchronization, and the heavy presence of delocalized plans to implement synchronization goals. We first investigated whether a *systematic* approach to comprehension correlates with success, as predicted in the classic paper by Littman *et al.* [10]. We found a correlation between systematic comprehension and success under some measures; however, the strategy is not a strong predictor of success. For instance, under one measure, only half of the participants who applied the strategy were successful.

Systematic comprehension aims to develop a *strong mental model* of the program. To understand where participants may have had difficulty achieving success with the approach, our study included a posttest to assess the strength of participants' mental models in terms of two kinds of

*This material is based in part upon work supported by LogicBlox Inc and 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 LogicBlox Inc or the National Science Foundation. Licenses for Camtasia provided by TechSmith Inc.

knowledge—*static* and *causal*. Participants were generally able to acquire the forms of static knowledge that are specific to concurrent software. For instance, they were largely able to understand key data and thread roles involved in synchronization and to articulate the lifecycles of objects and threads. Moreover, successful participants scored significantly higher on questions regarding these issues than did the unsuccessful participants. Far fewer were able to answer questions involving causal knowledge—for example, to identify when a thread executing in a method might block, to predict what could happen to shared data in between when a thread invokes a *wait* statement and when that invocation returns, and to reason about interaction with other agents. This weakness of the mental model in terms of causal knowledge may explain why the systematic strategy is not a strong predictor of success in concurrent software.

Finally, we sought to reconcile phenomena specific to concurrency with the concepts and predictions of Vessey's theory of fault diagnosis [17]. She claims that experts apply *breadth-first problem solving*, which involves pursuing multiple lines of reasoning and deliberately investing intellectual resources into competing hypotheses to avoid jumping to conclusions. Because experts should be more successful than novices, breadth-first solving should at least correlate with success. It stands to reason that the strategy would be especially important in the context of concurrent software, because failures can be difficult to reproduce reliably using testing, and the intellectual effort required to reason about thread interleavings is substantial. Using formal fault-tree analysis [4, 9] to enumerate a space of potentially relevant and competing hypotheses for the fault under study, we sought to understand how and how effectively the strategy was applied in the concurrency context. Participants applied the strategy to analyze hypotheses that require reasoning about sequential behavior, but they seemed to abandon it when analyzing hypotheses that require reasoning about thread interleavings. Consequently, many potential causes were never explored and perhaps never even considered.

2. Background and Related Work

The *think-aloud method* is a rigorous empirical technique used to obtain a model of the cognitive processes that take place during an activity or to test the validity of a proposed model [6, 16]. A *think-aloud protocol* is a transcript of the utterances of the participants as they are engaged in some activity. An associated *action protocol* describes what participants do as they engage in this same activity. Because the time and effort required from both participants and researchers for such intensive user studies is substantial, the number of participants is typically fewer than 20 (e.g., 5 for [19], 13 for [8], and 20 for [15]).

The literature contains a number of observational studies designed to investigate the factors and behaviors that lead to success when programmers perform maintenance on sequential systems. Littman *et al.* argue that a *systematic* approach to comprehension is more effective than an *as-needed* strategy [10]. The systematic strategy involves starting at the beginning of the program and tracing the flow of the entire program, using various forms of simulation.[1] The as-needed strategy involves studying only those portions of the code that are believed to be useful for the task at hand. Moreover, two distinct kinds of knowledge—*static* and *causal*—must be gained during systematic comprehension [10]. Static knowledge refers to an understanding of a program's functional components (e.g., roles, classes and methods), and causal knowledge is an understanding of how the functional components interact at run time.

Vessey conducted a well-known study that focused specifically on corrective maintenance tasks. She argues that experts tend to solve a debugging problem using a deliberate and precautionary strategy of hypothesis generation and validation [17]. This *breadth-first* approach involves first gaining a high level of understanding of the problem—in the process making hypotheses about the cause of the failure—and then attempting to verify or refute the hypotheses. By contrast, the *depth-first* approach involves attempting to verify hypotheses as they are formed—prior to gaining a high-level understanding of the problem. She found that expert participants used the breadth-first approach in conjunction with what she calls "system thinking," and novice participants used both breadth-first without system thinking and depth-first approaches.

Von Mayrhauser and Vans developed a model of program comprehension that combines the elements of several prior cognition models [18]. Their *integrated cognition model* explains how programmers comprehend programs during maintenance; however, to our knowledge, it does not prescribe which strategies and behaviors lead to success on task. Sillito, Murphy, and De Volder studied the information needs of programmers during maintenance to better inform tool design [14]. They created a catalog of questions programmers ask during maintenance. Because we are interested in how the presence of concurrency impacts success, we chose to focus on theories of comprehension which make specific predictions relating strategies and behaviors to success.

3. Method

Fifteen students in a graduate-level formal methods course participated in the study. All participants happened to be male.

[1]Robillard, Coelho, and Murphy refined these ideas for systems too large to be understood in their entirety [12].

3.1. The Program

Participants were asked to perform corrective maintenance of a small but representatively complex multi-threaded program into which we had seeded a fault. The program simulates an e-business server. The server accepts network connections from remote clients, receives requests from the clients over these connections, and simulates processing of the requests. The server uses multiple threads. A lone *listener thread* accepts client connections and places the requests received over these connections on a shared queue. Meanwhile, multiple *handler threads* contend for requests by synchronizing on the shared queue.

We seeded the server with a fault in the synchronization logic responsible for scheduling handler threads to dispatch requests when they arrive. This fault is representative of the class of synchronization-related faults that are difficult to reliably reproduce by running the program. It manifests in a failure only under certain timing and load constraints. To facilitate reproduction of the failure, we provided the participants a separate *stress tester* program, which issues client requests to the server at a user-adjustable speed, thereby allowing testing under various loads.

The seeded fault stems from the way the listener and handler threads synchronize their access to a shared queue of connection requests. This *request queue* is managed by an object called the *pool*, which encapsulates and synchronizes access to both the request queue and a buffer of handler objects, which are used to host the handler threads.[2] The pool defines a mutex lock for each resource that it manages and a host of queue/buffer-specific operations, each of which is implemented so as to acquire (and release) the appropriate mutex at the beginning (and the end) of the operation.

The request queue is accessed through operations submit_request and retrieve_request (Fig. 1). Notice that both methods are bracketed by calls to acquire and release on a mutex lock called queue_lock_. These invocations are necessary to guarantee the methods execute under mutual exclusion—that is, to ensure that multiple threads do not concurrently access the queue.

The *condition synchronization* logic in lines 17–20 of retrieve_request and lines 6–9 of submit_request blocks a thread that invokes retrieve_request until such time as the request queue contains a request to retrieve. The condition synchronization is orchestrated using the *condition variable* nonempty_queue_cond_. The listener thread signals this condition variable when it adds a request to an empty request queue if any handler threads are blocked waiting to retrieve requests from the queue. The variable queue_waiters_ records a count of the number of handler threads currently waiting for a request to be placed in the queue. The listener thread uses it to optimize the number of signals it sends—specifically to keep from signaling threads when none are waiting and to issue a sufficient number of signals when multiple threads are waiting.

```
 1 void Pool::submit_request(Request* request)
 2 {
 3   queue_lock_.acquire();
 4   request_queue_.push_back(request);
 5
 6   if (queue_waiters_) {
 7     nonempty_queue_cond_.signal();
 8     --queue_waiters_;
 9   }
10
11   queue_lock_.release();
12 }
```

```
13 Request* Pool::retrieve_request()
14 {
15   queue_lock_.acquire();
16
17   if (request_queue_.empty()) {
18     ++queue_waiters_;
19     nonempty_queue_cond_.wait();
20   }
21
22   if (request_queue_.empty()) {
23     queue_lock_.release();
24     return 0;
25   }
26
27   Request* request = request_queue_.front();
28   request_queue_.pop_front();
29
30   queue_lock_.release();
31   return request;
32 }
```

```
33 int Pool::dispatch_request()
34 {
35   // First, retrieve a handler from the pool.
36   Request_Handler* hdlr = retrieve_handler();
37
38   if (hdlr == 0) return -1;
39
40   // Second, retrieve a request from the queue.
41   Request* request = retrieve_request();
42
43   if (request == 0) return -1;
44
45   // Third, use handler to process request.
46   if (hdlr->process(request) == -1) return -1;
47
48   // Fourth, return the handler to the pool.
49   add_handler(hdlr);
50
51   return 1;
52 }
```

Figure 1. Relevant eBizSim source code.

[2]The pool object is so named because multiple handler threads pool up around it waiting for work to be dispatched to them. This design is sometimes described as an instantiation of the *reactor pattern* [13].

The seeded fault appears on line 17. The line *should* begin a while loop, but we replaced the `while` keyword with an `if`. To see how this fault may manifest in a failure requires reasoning about possible interactions between two or more handler threads and the listener thread when the request queue is empty. We should expect condition synchronization to be difficult to comprehend for several reasons. First, the logic required to implement it necessarily manifests as a *delocalized plan* [15]. Second, the wait/signal primitives have side effects on unseen OS-level resources (such as thread wait queues and mutex locks) and these interactions are difficult to reason about [20]. Finally, because unoptimized condition synchronization may incur an unacceptably large cost in terms of context switching, the logic is often highly optimized. Each of these characteristics bodes poorly for program comprehension.

3.2. Procedure

The study began with group instruction, in the form of a 50-minute lecture, on concurrency constructs and their implementation using the ACE wrappers toolkit [13]. The goal of this lecture was to ensure that participants were well-prepared to undertake the assigned maintenance task and to mitigate the effects of differences in prior knowledge on their performance. The participants then took a pretest on concurrency terminology and concepts. Only two of the participants scored below 50% on the pretest, and a majority (8) scored above 75%.

Individual 3-hour sessions were then conducted in a private office. Participants were provided with a workstation equipped with standard desktop and development software, and access to the eBizSim source code. We outfitted the workstation with a microphone headset and the *Camtasia* video capture software, which we used to capture the screen interactions and speech of the participants in the form of a video of the computer screen, with voice-over from the participant. During the first 15 minutes, participants were introduced to the equipment and environment, audio collection was calibrated, and participants engaged in think-aloud on a warm-up task. Participants were then given a brief tour of the directories containing the eBizSim software, provided with a bug report, and asked to perform corrective maintenance on the software.

Prompters trained in the think-aloud method accompanied the participants as they engaged in the maintenance task and, as needed, asked the participants to "please, keep talking." Fig. 2 depicts the bug report. It describes the output associated with the failure as well as some tips on how to reproduce the failure. Scratch paper, a brief guide to concurrency constructs in ACE, and a C++ manual were also provided. Participants were permitted to browse the Internet as they deemed necessary. They were allotted up to 150 minutes to complete the task. Following the sessions, participants took a posttest designed to evaluate program comprehension.

We are experiencing a problem with the eBizSim server program, wherein it intermittently exits with the error message:

```
error: Pool::dispatch_request() failed
```

This error has been fairly difficult to reproduce. So far, the most reliable way we have found to reproduce it is to run the stress tester with a setting of 4.27. Even with this setting, the program may take several minutes to exhibit the error. Occasionally, the program will run at the above setting for a long time (on the order of 5 minutes) without failing. In these cases, restarting the server and the stress tester seems to help in drawing out the error.

Figure 2. Bug report provided to participants.

4. Systematic Comprehension Strategy

We examined our think-aloud data to find evidence of participants applying a systematic approach to comprehension. By definition, the approach involves the programmer performing "extensive [global] symbolic execution of the data flow paths between subroutines" [10]. *Global symbolic execution* refers to starting at the main routine and following the control and calling structure of the subroutines. By virtue of this symbolic execution, the programmer must "actually imagine the behavior of the program as if it were running in time," thereby providing her with "causal knowledge about the order of actions in the program" [10]. Causal knowledge is needed to recognize and fix the fault in the eBizSim program, where faulty synchronization logic leads to an unintended ordering of the program actions under certain thread schedules. Other prior work extols the effectiveness of the systematic strategy for understanding programs with delocalized plans [15]. Synchronization logic typically manifests as a delocalized plan. Finally, as in the Littman study, the eBizSim program is small enough to be read in its entirety within the time frame of the participant sessions.

We analyzed our think-aloud data to learn (1) whether the systematic approach is applied and how frequently, and (2) whether its application correlates with success. We coded participants as using the systematic strategy if they made it a goal to first understand the entire unmodified program, before attempting to diagnose and correct the fault. Such participants investigated the code by starting from the main function and tracing the control-flow and calling paths, by reading the contents of each file from top to bottom, or by some combination of the two. We coded participants as using the as-needed strategy if they were clearly not concerned with understanding the entire program, but rather only wanted to understand enough to complete the task. Such participants investigated the program by start-

	01	02	04	13	14	06	08	09	10	12	07	15	03	05	11
Strategy	A	A	A	A	A	S	S	S	S	S	S	S	S	S	S
Fixed	+	-	-	-	-	+	+	+	+	+	+	+	-	-	-
NoNew	+	-	-	-	-	+	+	+	+	+	-	-	-	-	-
Explained	-	-	+	-	-	+	+	+	-	-	+	-	-	-	-

Table 1. Strategies and success.

ing at the point in the code where the failure manifested (as indicated by the bug report) and, using local information, examined only the parts of the code they felt they needed to understand. Based on this encoding, we determined that ten of the fifteen participants applied the systematic strategy. Table 1 depicts the results of our encoding in the row *Strategy*.

We measured success on task using three categories. The category Fixed records whether the participant found and successfully fixed the seeded fault—that is, he replaced the `if` with a `while` on line 17 in Fig. 1. Eight of the fifteen participants achieved this level of success (Table 1). A stronger measure of success is encoded in the category NoNew, which records whether, in addition to fixing the fault, the participant did not introduce any new faults. Only six of the participants achieved this level of success. Finally, the category Explained records whether the participant was able to successfully articulate the nature of the fault when answering the following question on the posttest:

> "Describe, in detail, the design fault that leads to the intermittent failure in the eBizSim server. You should be able to give a concrete scenario that demonstrates the fault."

This category is not necessarily either stronger or weaker than either Fixed or NoNew. In our study, only five of the fifteen participants achieved Explained.

Analysis of these data shows that participants who applied the systematic strategy were, on average, more successful on task by all measures of success. However, this result is statistically significant only in the case of Fixed ($p < 0.05$). In contrast to the results of the Littman study, our data show that use of the systematic strategy alone is not a strong predictor of success.

5. Static and Causal Knowledge

We next examined our think-aloud and posttest data to find evidence to evaluate the claim from Littman *et al.* of a correlation between success on a maintenance task and a strong mental model of the program, characterized by good static and causal knowledge about the program. We found significant differences between successful and unsuccessful participants for some types of both static and causal knowledge. Within the group who employed the systematic approach, we found similar trends, but these did not rise to the level of statistical significance within this small sample. Further, we found that participants overall had difficulty with certain types of questions. In the paragraphs below we present the types of static and causal knowledge we evaluated, examine the concepts that proved difficult for the participants, and discuss the elements of static and causal knowledge for which a significant difference was found between successful and unsuccessful participants. For purposes of analysis, we define *success* as in Section 4.

Qnum	**Overall**	**Fixed**	**NoNew**	**Explained**
1-S	82	88/76	89/78	**100/73** **
2-S	80	88/71	83/78	**100/70** *
3-S	77	84/68	**96/64** *	95/68
4-S	83	**100/64** *	**100/72** *	90/80
5.1-S	85	**100/68** *	**100/75** *	**100/78** *
5.2-C	55	53/58	58/53	70/47
6-S	88	**97/77** *	**100/80** **	**100/82** *
7-S	91	88/95	100/85	100/97
8-C	63	67/57	73/55	**89/48** **
9.1-C	79	86/69	**89/64** *	86/76
9.2-C	70	73/66	74/67	79/67
9.3-C	80	82/77	86/76	86/78
10.1-C	91	87/79	88/79	92/80
10.2-C	82	88/74	86/80	89/79
11-S	83	94/71	**100/72** *	80/85
12-S	87	88/86	83/89	80/90
13-C	33	25/43	17/44	40/30
14-C	67	**100/29** **	**100/44** **	**100/50** **
15-C	48	**72/20** **	**75/29** *	**90/26** ***
16-C	34	**55/71** *	57/19	**68/18** *
17-C	60	**94/71** ***	**100/33** **	**100/40** **

Table 2. Performance on posttest questions.

Static knowledge includes knowledge of the objects that the program manipulates, the actions the program performs, and the program's functional components—that is, segments of code that, together, accomplish a task. For multi-threaded programs, static knowledge includes knowledge of threads and their roles, shared data, and synchronization mechanisms. Knowledge of threads and thread roles includes knowing what threads are spawned by the program, what roles the threads play, and where in the code the threads are spawned, begin executing, and terminate. Knowledge of shared data includes knowing which data are shared by multiple threads, the roles of the threads that access the data, and the locations of critical sections involving the data. Knowledge of synchronization mechanisms includes knowing the mutex locks, condition variables, and abstract conditions that are used to synchronize accesses to shared data.

Littman *et al.* define causal knowledge as that pertaining to the interactions among functional components. The need to take into account the multitude of potential thread interleavings in a concurrent system makes understanding these potential interactions difficult.

Table 2 summarizes performance on the posttest, listing the overall average, and for each category of success, listing the average for the successful group and the unsuccessful group (S/U). Question numbers are annotated S (static) and C (causal) to indicate the type of knowledge the question is designed to evaluate. For each question, we performed a heteroscedastic, two-tailed Student's t-test to determine if the difference in means between the groups is significant. Significant differences are indicated in bold-face, with $* = (p < 0.05)$, $** = (p < 0.01)$, and $* * * = (p < 0.001)$.

Participants performed well when evaluating static knowledge of important objects in the implementation of the eBizSim server (Q1). A significant difference was found only on the Explained categorization.

Participants also performed well on questions that evaluated knowledge of shared data, addressing shared objects and data structures (Q4, Q5.1), scoring over 80% when asked to identify server classes that might be concurrently accessed by multiple threads (Q4) and when asked to describe the purposes of the main data structures (Q5.1). Significant differences were found on all categorizations of success for question 5.1 and for all but Explained for question 4. However, when asked to identify the different types of threads (i.e., thread roles) that might be involved in concurrent access to those data structures (Q5.2) performance dropped to 55%. This drop was seen across both the successful and unsuccessful groups.

Questions 2, 3, 11, and 12 evaluated static knowledge of threads and thread roles. Participants were generally able to propose names for the two roles that threads could play and to describe the responsibilities assumed by threads playing each role (Q2), to state how many threads might play each role (Q3), and to answer questions about thread creation (Q11,Q12). Performance on these questions correlated with some categorizations of success and not with others, but with no clear pattern.

Participants were successful when asked to describe the "life cycle" of the Request_Handler objects in the system (Q6). These objects are created when the system initializes, used to process requests during their lifetime, and then destroyed when the system terminates. Differences between successful and unsuccessful participants were significant in all three categories.

Questions 7 and 8 addressed the behavior of the dispatch_request method, asking participants to list the major activities performed during this method (Q7) and to identify those activities in which the actor might block and explain the conditions and synchronization objects and operations involved (Q8). Participants were able to list the major activities, but no significant difference existed across the groups. However, they struggled to identify all of the conditions under which an actor might block, with many participants either stating that an actor could block trying to acquire the mutex lock, or that an actor could block on a condition variable, but failing to state both. The ability to identify these conditions was significant only for the Explained categorization.

Questions addressing causal knowledge proved challenging. Two of these causal questions (Q9 and Q10) were scenario based. Such questions posit some state of the system (e.g., "... a thread, call it T, after beginning execution of dispatch_request, successfully retrieves a request handler from the pool"), and then ask what may occur next if, for example, the queue is empty, or has 1 request, or has 10 requests. A list of six activities was provided, and the participants were asked to select those activities that could occur next. Two participants misunderstood the questions and were removed from consideration in evaluating the effects of these questions. Of those remaining, a statistically significant difference was found only for the NoNew categorization, and only for Q9.1. Although participants performed well on these questions overall, a sub-group of scenarios proved problematic (average score 46%). All of the problematic scenarios involved reasoning about how the state of the system may change between the time a thread invokes a *wait* statement and when that invocation returns. Condition synchronization is generally difficult to reason about because so much may transpire between when a thread invokes and returns from a wait. Many of the incorrect answers we observed in questions related to condition synchronization indicate that participants made incorrect inferences about causal relationships between the values of counting variables (such as queue_waiters_) and the number of threads currently blocked on the queue.

Questions 14 and 15 honed in on the actual fault, asking participants to identify the nature of the synchronization fault that was seeded into the program. Q14 was a multiple-choice question, in which the correct description of the fault was selected by 67% of the participants. Q15 asked participants to describe in detail and in their own words " the design fault that leads to the intermittent failure in the eBizSim server." Only four participants received full credit for this question, despite greater numbers of participants who were able to find and fix the flaw, or to select the correct description from a multiple-choice question. Clearly, this causal knowledge and the ability to articulate it is more difficult than identifying the fault from a series of choices or than actually fixing the fault in the implementation. Note: A correct answer to Q15 served as the basis for inclusion in the successful group for Explained.

Participants struggled when asked (Q16) to consider interactions of the eBizSim server with an outside agent, the stress tester, and to discuss the effect of stress-test speed in causing the server to crash. Significant differences between the successful and unsuccessful groups were found under both the Fixed and Explained categorization.

Finally, participants were asked to describe how they fixed or would fix the design flaw that leads to the intermittent failure in the eBizSim server (Q17). Interestingly, participants performed better (60%) at describing the fix than at describing the fault. A significant difference between the successful and unsuccessful groups was found under all categorizations.

In summary, participants were generally able to acquire static knowledge relevant to a multi-threaded program, performing well at understanding key data structures and thread roles involved in synchronization and describing the lifecycles of objects and threads. They began to struggle when asked to list all of the conditions under which an actor might block, often recalling only some of those conditions (mutex locks or condition variables). Further, they had greater difficulty with questions related to causal knowledge, particularly with certain scenario-based questions, and with free-form explanations of the nature of the fault itself and of interactions with other agents.

Littman *et al.*'s claim of a correlation between strong mental models and success on a maintenance task appears to hold here. Types of knowledge that appear to be related to success, under one definition or another, included static knowledge of shared objects and data, and detailed knowledge of the life cycles of threads and objects. Certain types of causal knowledge were seen to have a significant effect on success, including, not unexpectedly, the ability to describe concrete scenarios under which a fault might occur. Of further interest is the nature of the scenario-based questions that proved most difficult to participants. How these difficulties might be remediated through tools or educational approaches is an open question.

6. Breadth-First Fault Diagnosis

Because errors in a concurrent program are not predictable, analysts should consider multiple competing hypotheses when diagnosing a fault. However, reasoning about thread interleaving is complex. Thus, analysts should invest effort in such activities sparingly and deliberately. Vessey refers to such behavior as *breadth-first* problem solving and suggests that experts apply it during corrective maintenance [17]. We investigated the application of this strategy in our study. Specifically, we studied how well participants generated hypotheses and whether they appeared to be considering multiple competing hypotheses rather than jumping to conclusions. To support this analysis, we developed a *fault tree* [4, 9] for the failure in our study (Section 6.1). By associating verbalizations/actions in the protocols to events and patterns of elaboration in the fault tree, we observed: (1) a subset of participants seem to have been applying breadth-first solving, (2) membership in this group correlates with success on task, and (3) key hypotheses went unexplored. (Section 6.2).

6.1. Fault Tree

Fig. 3 depicts a portion of the fault tree for the failure under study. Each box represents an *event*, which is an observation at some moment in time. Observations refer to properties of system objects and are made at distinct *control points* in the program. We distinguish three kinds of control points, formalized in terms of control flow graphs (CFG).[3] Namely, for a thread T and CFG nodes N and N', an observation may be made (1) when T enters N, denoted T enters N; (2) when T exits N, denoted T exits N; or (3) when T traverses the CFG edge from N to N', denoted T trav $N \rightarrow N'$. By convention, T enters N occurs prior to when T actually executes the instruction at node N, which means another thread could execute between the observation and T's execution; whereas T exits N is coincident with termination of the instruction at N.

We designate CFG nodes informally using the concrete syntax of a program statement or expression (e.g., $[\![$`accept_request()`$]\!]$) or the line number upon which it appears. Where necessary, we prefix an informal designator with a *qualifier* to indicate the type of instruction in the CFG node being designated. For instance, call(41) designates the node containing the actual `call` instruction as distinct from the other CFG nodes a compiler might generate for this assignment statement whose right-hand side includes a method invocation. Qualification allows an observation to refer to the value returned from the call. Likewise, ifCond designates the node containing the `if-goto` instruction that branches to the CFG node designated by the target of a trav specifier.

In addition to designating a control point, each event asserts a state predicate and an optional timing constraint. A timing constraint specifies when the event occurred relative to other events. For expressing timing constraints, we use $M.t$ to denote the time event M occurred.

The root event in Fig. 3 represents an error state that manifests in the failure described in Fig. 2. The error occurs at time $root.t$ when a thread T_0 observes an invocation of the dispatch_request method returning -1. The state predicate, which appears below the control point designator, uses the keyword *rval* to refer to the value returned from the designated `call` instruction.

The children of an event represent events that enable (or contribute to enabling of) the event. Or-gates indicate alternative enabling events; whereas and-gates indicate a conjunction of multiple events, all of which must occur to enable the parent. For example, the code for dispatch_request can return -1 in any one of three statements. Nodes L_1, M_1, and R_1 represent events that will

[3]Here, we assume CFG nodes are labeled with instructions in three-address code [1]. A statement in the program may therefore engender many nodes and edges in the CFG.

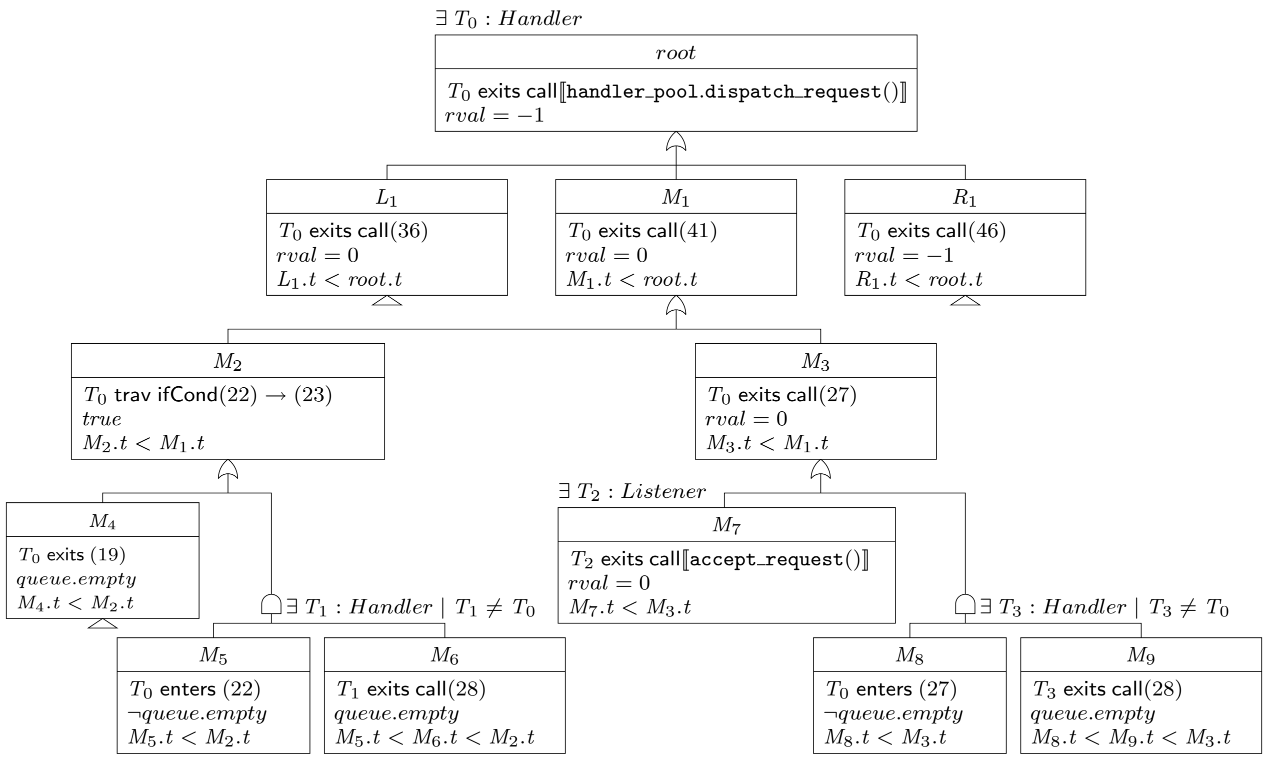

Figure 3. Fault tree.

cause one of these statements to be executed. Evaluation of retrieve_handler at line 36 could return 0 (L_1), thereby causing dispatch_request to return 0 at line 38. Nodes M_1 and R_1 can be understood similarly. Due to space limitations, we show only part of the M_1 subtree, which is the alternative that actually caused the seeded fault. A triangle indicates that a subtree has been elided.

The M_1 tree explains how an invocation of retrieve_request could have returned 0. Either evaluation of the if condition on line 22 succeeded (event M_2), or the method invocation on line 27 returned 0 (M_3). Likewise, the M_2 tree explains how the if condition on line 22 could have succeeded. Either T_0 exits the body of the first conditional block with the queue being empty (M_4); or T_0 reaches line 22 believing the queue is non-empty, but another thread empties the queue before T_0 can execute the if statement on line 22. The remaining events in Fig. 3 model how T_0 could have retrieved a 0 that was inserted into the queue by the listener thread (represented by T_2) or how T_0 could have tried to pop an empty queue.

6.2. Fault-Tree Coverage

We used the fault tree to help identify those participants who applied breadth-first solving and also to identify relevant hypotheses that were overlooked, perhaps due to the complexity of reasoning about concurrent phenomena. We say that a participant *discovers* an event if he somehow verbalizes the control point, state predicate, and/or time constraint associated with the event; whereas he *analyzes* the event if he attempts to determine how it is enabled. In some cases, event discovery and analysis are clearly articulated in the protocols. For instance, **par06** was analyzing the root event when he uttered, "under what conditions can we end up returning an error from dispatch request?" While trying to answer this question, he discovered and succinctly verbalized the enabling events L_1, M_1, and R_1. In other cases, we had to infer event discovery and/or analysis from clues in the protocols. For instance, we coded **par09** as having discovered event M_4 when, while analyzing event M_2, he verbalized a scenario in which a waiting thread awakes to find the request queue empty. Because it was discovered during the analysis of M_2, we say **par09** discovered M_4 as an enabler while analyzing M_2.

Table 3 depicts the events from the *root* and M subtree that each participant attempted to analyze. We omitted the L and R subtrees from the table to save space and because their data are consistent with those of the M subtree. The far right column indicates the percentage of events in the M subtree that participants discovered. The bottom row provides the sum of each column.

	root	M1	M2	M3	M4	M5	M6	M7	M8	M9	M-Pct
01	+	+	+	+	+	-	-	-	-	-	44%
02	+	+	-	+	-	-	-	-	+	+	44%
03	+	+	+	-	-	-	-	-	-	-	22%
04	+	+	+	+	+	+	+	-	-	-	67%
05	+	+	-	-	-	-	-	-	-	-	11%
06	+	+	+	-	+	-	-	-	-	-	33%
07	+	+	+	-	+	-	-	-	-	-	33%
08	+	+	+	-	+	-	-	-	-	-	33%
09	+	+	+	-	+	-	-	-	-	-	33%
10	+	+	+	-	+	-	-	-	-	-	33%
11	+	+	+	-	+	-	-	-	-	-	33%
12	+	+	+	-	+	-	-	-	-	-	33%
13	+	+	+	-	+	-	-	-	-	-	33%
14	+	+	+	-	+	+	+	-	-	-	56%
15	+	+	+	-	+	-	-	-	-	-	33%
Total	15	15	13	3	12	2	2	0	1	1	

Table 3. Fault tree coverage.

These data show that participants generally analyzed certain events (e.g., M_1, M_2, and M_4), and generally ignored or failed to discover others. Every participant analyzed $root$ and M_1. Within M_1, 13 participants analyzed M_2, and only 3 analyzed M_3. Within M_2, 12 analyzed M_4, and only 2 analyzed the M_5/M_6 subtree. Within M_3, no one analyzed M_7, and only 1 participant analyzed the M_8/M_9 subtree. No participant analyzed every event in the M subtree, and only 2 participants analyzed at least half of the M events.

We identify participants who are likely to have been applying breadth-first solving using transcript data and verifying that the participants analyzed a sufficiently large number of the events in the fault tree down to depth 2. This collection of events comprises $root$, L_1–L_3 (not shown in figure), M_1–M_3, and R_1–R_2 (not shown in figure). This group contains ten participants if we exclude events L_3 and M_3 as outliers.[4] Only **par01** analyzed all events down to depth 2 in the fault tree. Seven participants within this group were successful in diagnosing the fault and 3 were not. Only one successful participant was not in this group.

Focusing on the group we coded as having applied breadth-first solving, nine of the ten analyzed events L_4 and M_4. No one in the group considered events L_5–L_9 and M_7–M_9, and only 1 considered the M_5/M_6 subtree. These omissions are noteworthy. Perhaps the participants ran out of time. Although it seems unlikely that they all ran out of time at nearly the same points, they may have grown weary after working for over two hours and begun to make errors of omission. Alternatively, given the rate of success in this group, they may have decided that they had found and fixed all of the faults in the system. However, if they drew this conclusion without having analyzed a large swath of potential causes of failure, then either: (1) we wrongly included them in the group applying a breadth-first solving strategy, (2) at some point during the study they abandoned this strategy, or (3) something about the problem made the omitted hypotheses difficult to formulate. Assuming our grouping is correct, we interpret these omissions as a breakdown of the breadth-first solving strategy, which should otherwise have led to the discovery of these events.

[4]only one participant in this group analyzed L_3, and only three analyzed M_3.

To see whether concurrency was a factor in this breakdown of strategy, we looked at which nodes require reasoning about sequential behavior only and which also require reasoning about concurrent behavior. For example, determining that M_1 enables $root$ involves reasoning only about the sequential control flow in the dispatch_request method. In contrast, determining that M_7 enables M_3 requires reasoning about the concurrent behavior of two threads (T_0 and T_2 in Fig. 3). The nodes in Table 3 that require reasoning only about sequential behavior are $root$, M_1, M_2, and M_3. The rest also require reasoning about concurrent behavior.

In our data, hypotheses that require reasoning about concurrent behavior are less likely to be analyzed than those involving sequential reasoning. In the concurrent category, 5 of the 6 events were analyzed by 2 or fewer participants, and only 1 (M_4) was widely analyzed. In the sequential category, 3 of the 4 nodes were analyzed by 13 or more participants, and only 1 (M_3) was largely unanalyzed. Moreover, the code that gives rise to this lone unanalyzed event (M_3) follows (in the source code) from the code associated with M_2, which contains synchronization primitives. When concurrent reasoning is required, participants seem largely to pursue a small number of hypotheses and fail to consider the competitors. This suggests concurrency may be a factor in the breakdown of the breadth-first solving strategy.

7. Conclusions and Open Questions

Participants in our study overwhelmingly applied a systematic approach to comprehension to diagnose the fault. Unfortunately, even though use of the strategy correlated with success on task, it was not a strong predictor. In terms of developing strong mental models of a program, we identified several types of knowledge that are specific to concurrent software and that appear to be related to success. Most notably, static knowledge pertaining to the where in the code threads and objects are created and destroyed and certain forms of causal knowledge have a significant effect on success. That static knowledge would have an effect was somewhat surprising but is encouraging because it might be relatively easy to develop tools and/or reading techniques that improve facility in its development.

How to support acquisition of the more difficult forms of causal knowledge is an open problem. One strategy for coping with its complexity is to formulate *invariant properties* which are satisfied by all action orderings in a system [2]. For instance, many multi-threaded objectoriented programs are designed so that shared objects execute method invocations with *monitor semantics* [11]. Given such a program and a strong *monitor invariant*, the programmer might be able to reason more effectively about the scenario-based questions and the kinds of design faults that we seeded in this study. Perhaps a programmer could systematically identify the monitor objects in the system and their monitor invariants. Such an approach could be supported by prescriptive guidance in the form of a structured reading technique [3].

Our studies also showed a breakdown in what was otherwise an orderly, breadth-first analysis of the fault. Breakdown seemed to be triggered by hypotheses involving thread interleavings. In our previous study, we observed participants proceeding with incomplete causal knowledge and investing heavily in what we refer to as *failure-trace modeling* [7]. Failure-trace modeling involves a heavy investment of mental energy and is definitely more of a depth-first than a breadth-first strategy. Whether tools or prescriptive guidelines can be developed to prevent breakdowns of breadth-first solving in this context is an open question. With respect to tools, we are investigating automated approaches to elaborating the fault events and producing trees such as in Fig. 3. It is also conceivable that a lightweight tool—for example, a tool for visualizing and recording the fault-tree structures—could help to avert breakdown. It may also be that the space of hypotheses one would need to generate to diagnose a fault in a concurrent program is so large that a breadth-first approach may become intractable except for very small programs. How to remediate this situation is an open research question.

Finally, we recognize several threats to validity in this work. The first concerns how well our seeded fault represents the kind of synchronization faults that arise in practice. Other concerns include the limited scale of both the program and the change activities, and the absence of strong incentives for participants to succeed. Such problems are difficult to address given the nature of a think-aloud study, as participants can be asked to participate for only a relatively limited length of time. In the future, we will develop other kinds of studies (e.g., case studies) to validate our results on programs and change activities of larger scale and with a more realistic structure of incentives. Another potential threat to validity concerns composition of the student pool, all of whom were male. The use of the think-aloud method is a potential threat to validity because the cognitive resources required for introspection may affect how participants perform. However, numerous studies show that participants who are asked merely to “verbalize their inner dialogue”, as were the participants in this study, perform comparably on measures of performance with participants who were not asked to think aloud [5].

References

[1] A. Aho et al. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, second edition, 2007.

[2] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[3] V. Basili et al. Studies on reading techniques. In *Proc. 21st Annu. Software Eng. Workshop*, 1996.

[4] S. J. Clarke and J. A. McDermid. Software fault trees and weakest preconditions: a comparison andanalysis. *Software Eng. J.*, 8(4):225–236, 1993.

[5] K. A. Ericsson. Valid and non-reactive verbalization of thoughts during performance of tasks. *J. Consciousness Stud.*, 10(9–10):1–19, 2003.

[6] K. A. Ericsson and H. A. Simon. *Protocol Analysis: Verbal Reports as Data*. MIT Press, 1993.

[7] S. D. Fleming et al. A study of student strategies for the corrective maintenance of concurrent software. In *Proc. IEEE Int. Conf. Software Eng.*, 2008.

[8] T. D. LaToza et al. Program comprehension as fact finding. In *Proc. 6th ESEC/SIGSOFT Symp. Foundations Software Eng.*, 2007.

[9] N. G. Leveson, S. S. Cha, and T. J. Shimeall. Safety verification of Ada programs using software fault trees. *IEEE Software*, 8(4):48–59, 1991.

[10] D. C. Littman et al. Mental models and software maintenance. *J. Syst. Software*, 7(4):341–355, 1987.

[11] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, second edition, 2007.

[12] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Software Eng.*, 30(12), 2004.

[13] D. Schmidt and S. D. Huston. *C++ Network Programming: Mastering Complexity with ACE and Patterns*, volume 1. Addison-Wesley, 2002.

[14] J. Sillito, G. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proc. ACM SIGSOFT Symp. Foundations Software Eng.*, 2006.

[15] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11), 1988.

[16] M. W. van Someren, Y. F. Barnard, and J. A. C. Sandberg. *The Think Aloud Method*. Academic Press, London, 2004.

[17] I. Vessey. Expertise in debugging computer programs: a process analysis. *Int. J. Man-Mach. Stud.*, 23(5), 1985.

[18] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Trans. Software Eng.*, 22(6), 1996.

[19] C. Wallace et al. Assertions in end-user software engineering: a think-aloud study. In *Proc. IEEE 2002 Symp. Human Centric Comput. Lang. and Env.*, 2002.

[20] S. Xie, E. T. Kraemer, and R. E. K. Stirewalt. Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. In *Proc. IEEE Int. Conf. Software Eng.*, 2007.