This is to certify that the
dissertation entitled

SUCCESSFUL STRATEGIES FOR
DEBUGGING CONCURRENT SOFTWARE:
AN EMPIRICAL INVESTIGATION

presented by

SCOTT DOUGLAS FLEMING

has been accepted towards fulfillment
of the requirements for the

Doctoral degree in Computer Science

R. E. K. Stirewalt
Major Professor's Signature

May 14, 2009
Date

SUCCESSFUL STRATEGIES FOR DEBUGGING CONCURRENT SOFTWARE: AN EMPIRICAL INVESTIGATION

By

Scott Douglas Fleming

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Computer Science

2009

ABSTRACT

SUCCESSFUL STRATEGIES FOR DEBUGGING CONCURRENT SOFTWARE: AN EMPIRICAL INVESTIGATION

By

Scott Douglas Fleming

Concurrent software can provide substantial performance benefits; however, such software is highly complex. This complexity makes debugging concurrent software especially difficult. Debugging techniques that are highly effective for debugging sequential software are rendered ineffective by concurrency. Moreover, the literature provides little advice for programmers on what techniques are effective. To address this problem, we conducted in an empirical investigation to understand the strategies and practices that successful programmers use during debugging. Specifically, we carried out an exploratory study in which we observed fifteen programmers individually performing a debugging task on a multithreaded server application, which we seeded with a defect. To better understand the programmers' goals and intentions, we prompted them to "think aloud" as they worked. This study produced several promising theories. To test and refine one of these theories, we followed up the exploratory study with a controlled experiment. Three main claims emerged from our studies. First, programmers who are successful at debugging concurrent software use a previously-undocumented failure-trace modeling strategy, which involves modeling interactions among multiple threads to understand the potential behavior of a concurrent program. Second, the use of external representations, such as UML sequence diagrams, during failure-trace modeling enhances success with the strategy. Third, concurrency thwarts efforts to systematically manage hypotheses regarding the cause of the defect during debugging. In this dissertation, we also report ancillary findings regarding other behaviors that programmers exhibited and share important lessons learned in the conduct of think-aloud studies.

To Grandpa Fleming, who waited a long time for this.

ACKNOWLEDGMENTS

It is my great pleasure to thank the colleagues, collaborators, friends, and family who supported me throughout my doctoral studies and who helped make this dissertation possible.

I would like to give special thanks to my PhD guidance committee members for providing me with expert counsel and thoughtful feedback. Words cannot express how grateful I am to my PhD advisor, R. E. Kurt Stirewalt, for his patience, superlative advice, and unwavering belief in me throughout this process. I give heartfelt thanks to Laura Dillon and Eileen Kraemer for going far above and beyond the call of duty as mentors and research collaborators. Many thanks to D. Zachary Hambrick and John Hale for their exemplary service as committee members and for the unique perspectives they brought to the group.

I would like to thank the individuals who assisted me in carrying out my studies. Thanks to Shaohua Xie for his hard work analyzing data for my exploratory study. Thanks to the undergraduate volunteers who served as prompters and transcribers during my exploratory study. Thanks to Alex Liu and Yi Huang for their assistance in conducting my controlled experiment. I would be remiss if I did not thank the individuals who, in good faith, participated in my studies and provided me with a window into the inner workings of programmers.

Thank you to the students, faculty, and staff of the Computer Science and Engineering Department who have been wonderful companions and contributed tremendously to my personal and professional growth. I give special thanks to my friends from the Software Engineering and Network Systems Laboratory: Ben Beckmann, Brian Connelly, Matt McGill, Chad Meiners, Andres Ramirez, Jesse Sowell, and everyone else!

Finally, I most want to thank my smart, dynamic, patient, thoughtful, and beautiful wife, Denise Fleming. Her constant support has kept me going through the ups and downs of this long, arduous journey. My dear, I cannot wait to begin the next phase of our lives together!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 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 have well-understood, highly-effective techniques to debug sequential (i.e., non-concurrent) software. Unfortunately, concurrency renders these techniques ineffective. Moreover, the literature is largely silent on what techniques are effective for debugging concurrent software. Therefore, we seek to understand the strategies that successful programmers use during the debugging of concurrent software and to develop tools and techniques that support and improve those strategies.

Programmers depend on two core techniques for debugging sequential software: *cyclic debugging* and *dependence analysis*. In cyclic debugging, the programmer replays a failing run of the program repeatedly while observing different aspects of the program's internal state [66]. Understanding the internal behavior of the program facilitates diagnosis of the defect. Unfortunately, failures of concurrent software may be difficult to reproduce, rendering cyclic debugging ineffective. In dependence analysis, the programmer traces data and control dependences backward through the code to find the source of an error [130]. Unfortunately, concurrent software exhibits a combinatorial explosion of data and control flows. This complexity makes data and control dependences difficult to identify and understand, rendering dependence analysis ineffective. Others (e.g., [62, 66, 86]) have tried to address these problems with tools; however, no specialized debugging tools have seen widespread use in practice [74].[1]

Our work aims to understand the strategies and practices that successful programmers use in debugging concurrent software. The literature contains numerous studies of the behaviors programmers exhibit during debugging (e.g., [46, 58, 123, 127]) and program comprehension (e.g., [65, 73, 100]), an activity closely related to debugging. This work has uncovered several strategies that predict success on such tasks. For instance, one seminal study [73] found that a *systematic comprehension strategy*, which involves tracing through the flow of control of the entire program prior to making modifications, leads to success on tasks involving unfamiliar code. Another seminal study [123] found that a *breadth-first approach*, which involves pursuing multiple lines of reasoning and deliberately investing intellectual resources into competing hypotheses regarding the cause of an error, leads to success on debugging tasks. However, none of this prior work specifically addresses concurrent software; therefore, it is an open question whether the prior findings will hold up in this context.[2] The research methods employed most commonly by this prior work emphasize observing programmers as they perform tasks, and collecting and analyzing predominantly qualitative data. To reveal the goals and intentions of participants, studies frequently ask them "think aloud" [39] as they work. We adopt methods that are consistent with these prior studies to address our research questions.

We began our investigation with an exploratory study in which we observed programmers debugging a multithreaded server application [42, 43]. This *exploratory research* emphasizes the generation of tentative hypotheses. We collected extensive observational data about their performances, which includes over 28 hours of video. This type of rich qualitative data is well-suited for exploratory research [37]. Like previous studies, we asked participants to think aloud as they work. Our analysis produced several promising

[1] Traditional parallel debuggers [77] are somewhat common, but these rudimentary tools fail to address the essential problems of concurrency.

[2] Others (e.g., [9, 52]) have empirically studied developers of concurrent software; however, their studies did not look at the detailed behaviors of programmers as they perform tasks.

hypotheses. Next, we conducted a follow-up experiment to test one of our hypotheses. Such experiments serve to increase confidence in the validity of hypotheses produced by exploratory research [8].

Based on the results of these studies, we make three central claims. Our first claim is that programmers who are successful at debugging concurrent software use a *failure-trace modeling* strategy. We were the first to document this strategy. Given a hypothetical error state of the program, the strategy aims to produce a *failure trace*—that is, a demonstration of how the system may transit among various internal states to arrive at the error state. It involves modeling scenarios of interactions among various threads in the system. Our exploratory study found that successful programmers were significantly more likely to use the strategy than unsuccessful. However, we also observed some limitations of the strategy. Specifically, a small number of participants were completely unsuccessful with the strategy, and a larger number, although successful on task, were unable to produce an actual failure trace.

Our second claim is that the use of external representations during failure-trace modeling enhances the success of the strategy. Such representations may take the form of diagrams, such as UML sequence diagrams. We believe that their use during failure-trace modeling reduces cognitive strain, making programmers less prone to mental errors. External representations in general have been shown to provide such benefits during cognitively-intensive tasks [139]. This claim emerged from the findings of our controlled experiment. We found that participants who externalized were significantly more able to reason correctly about scenarios of thread interaction than those who reasoned exclusively internally—that is, "in their heads." Such reasoning is key to success with failure-trace modeling. Participants in the exploratory study predominantly did their modeling internally. Therefore, we believe that cognitive strain was an important factor in the cases that were unsuccessful with failure-trace modeling.

Our third claim is that concurrency thwarts efforts to systematically manage hypotheses. For programmers to debug successfully in practice, they must systematically manage their hypotheses regarding the cause of the failure [123]. Systematic management involves the consideration of competing hypotheses and the careful tracking of which hypotheses have been investigated and which have not. The complexity of concurrent software typically leads to a large space of competing hypotheses. Forgetting to investigate hypotheses or having to reinvestigate hypothesis because the outcome was forgotten can have a dramatically negative impact on debugging. Although the program in our exploratory study was small, there were a fairly large number of plausible hypotheses as to the cause of its failure (roughly twenty). We found that many participants appeared to systematically manage hypotheses initially. However, we consistently observed a breakdown in the approach, and many plausible hypotheses failed to be investigated. Although the breakdown was not fatal in many cases, there is little doubt that it would be problematic in the context of large production systems with orders of magnitude more plausible hypotheses. It is an open question as to what caused this breakdown. However, as with failure-trace modeling, participants did little externalizing; therefore, cognitive strain is a promising suspect.

Our studies also produced several ancillary findings. For example, we found that participants universally engaged in cyclic debugging despite the well-known weaknesses of the technique. Not surprisingly, we found no relationship between the use of the technique and success. However, we did find that participants who used the technique without also engaging in failure-trace modeling were all unsuccessful. As another example, we found that the use of the systematic comprehension strategy did not lead to success. We believe that the strategy relies heavily on an implicit dependence analysis, which is known to be difficult with concurrent software. As a final example, we found that tweaking the code in the hopes of lucking into a fix is highly ineffective. Unsuccessful participants were significantly more likely to exhibit this behavior than successful.

In addition to research contributions of our work, we contribute lessons learned in the conduct of this type of research. For instance, the sharing of think-aloud data is problematic when studying programmers. In reporting on a think-aloud study, sufficient think-aloud data must be provided so that other researchers can replicate your analysis or run their own analyses [89]. Traditionally, textual listing of the participant utterances were sufficient because participants were working largely internally. However, programmers work with a rich set of external tools and documents. Utterances alone do not provide enough information to understand what they are doing. To make matters worse, attempting to document their actions is problematic because such documentation is highly subjective and may omit actions that are important to other researchers. A naive solution this problem is to share video data of participants. However, protecting the identities of participants is a foremost ethical concern [3]; photos of participants or audio of their voices should not be shared under any circumstances. To address these problems, we developed a method for sharing think-aloud data of programmers based on capturing computer-screen video and subtitling the video with participant utterances.

In addition to the above claims, our studies also generated new research questions to pursue. One question arises because most participants in our exploratory study were able to localize the failure to a particular segment of code. Two features of the study probably made this possible. First, participants were given enough information to locate a particular line of code that exhibited the failure. Although this line was far away from the defective line of code, it gave participants a clear starting point for their search. Second, they were able to reproduce the failure, although it tended to be difficult and time consuming. This feature enabled the limited use of cyclic debugging. However, not all failures in practice have such features. It is an open question whether failure-trace modeling would be as effective if localizing the failure was more difficult. Another open question is to what extent programmers have difficulty formulating hypotheses regarding the cause of a defect. We found that participants were less likely to analyze hypotheses that involve reasoning about multiple interacting threads than hypotheses that only involve a single thread. This suggests that programmers may have difficulty identifying certain types of hypotheses.

Future work should investigate the extent of this problem.

The remainder of this dissertation is organized as follows. Chapter 2 provides background on several relevant topics, including the related work on debugging concurrent software (Section 2.1), the empirical methods we use for our investigation (Section 2.2), the related empirical studies from the literature (Section 2.3), notations for externalizing models of thread interactions (Section 2.4), and the analysis of interaction complexity (Section 2.5). Chapters 3 and 5 describe the planning and analysis of our exploratory study, respectively. Chapters 6 and 7 describe the planning and analysis of our controlled experiment, respectively. Chapter 8 provides a discussion of our results, including their limitations and applications to the practice of software engineering. Finally, Chapter 9 summarizes our conclusions.

CHAPTER 2

BACKGROUND

Our work seeks to ameliorate the difficulty of debugging concurrent software by understanding the strategies and practices that successful programmers use to accomplish such tasks. The problem of debugging concurrent software is well-established and has consistently resisted attempts to solve it. Section 2.1 provides background on the nature of the problem and surveys the prior work on it. In particular, this prior work emphasizes the development of tools; however, we take a different approach, seeking to pursue solutions based on an empirically-grounded understanding of what works in practice. Section 2.2 describes the empirical methods that we will use to achieve this understanding. Others have applied similar methods to the study of software engineers. Section 2.3 surveys these related empirical studies.

In the course of our work, we developed two reusable technologies, which we offer here as contributions. The first contribution arose from our investigation of the benefits of externalizing representations of program behavior. We found that existing notations were inadequate for representing the behaviors of multithreaded software. Section 2.4 describes these notations and how we extended one to represent the behaviors of multithreaded software. Our second contribution arose from the need to evaluate the complexity of different program behaviors. The literature offers no metrics on which to base such evaluations. Section 2.5 describes several metrics we developed for this purpose.

2.1 Debugging Concurrent Software

Debugging concurrent programs is notoriously difficult. In Section 2.1.1, we describe two universal techniques that have traditionally been effective for debugging sequential software. In Section 2.1.2, we describe properties of concurrent programs that render these techniques ineffective. In Section 2.1.3, we survey the literature for prior attempts to alleviate this problem.

2.1.1 Debugging Sequential Software

Debugging is the process of diagnosing and correcting a defect in a program, which is known to be buggy. In this dissertation, a program *error* refers to a runtime behavior that violates the program's requirements. Not all errors are observable to the user—that is, the user may not experience a loss of functionality as a result of the error. We refer to losses of functionality as *failures*. The underlying cause of an error is a *defect* in the program code [138].

Diagnosing the defect accounts for the majority of effort during debugging [82]. The literature commonly characterizes diagnosis as a hypothesis-driven process [5, 46, 137, 138]. Following Araki and colleagues [5], the programmer maintains a working set of hypotheses about the cause of the error. As diagnosis proceeds, the programmer iteratively selects a hypothesis from the working set, verifies the hypothesis, and updates the working set based on the outcome of verification. The primary purpose of verification is to test the hypothesis—that is, to attempt to determine whether the hypothesis is correct. In the process of verifying a hypothesis, the programmer may make new observations about the program that are relevant to diagnosis. Based on the outcome of verification and any incidental observations, the programmer may modify the working set by generating new hypotheses, or by *authenticating* or *refining* existing hypotheses. Authentication involves deciding whether a hypothesis is correct or not. Deciding that one hypothesis is correct may imply that others are incorrect. Refinement involves making a hypothesis more constrained or more detailed. This diagnosis process proceeds until a hypothesis, which has been authenticated as correct, is sufficiently refined to explain the defect.

In the context of sequential programs, programmers universally employ two techniques to successfully diagnose defects: cyclic debugging and dependence analysis. These techniques are useful for verifying hypotheses and for reducing the space of plausible hypotheses.

Cyclic Debugging

Cyclic debugging involves observing selected properties of the program's internal state during a failing run [66]. The technique is *cyclic* because the programmer repeatedly reproduces the failing run, iteratively refining which properties he observes. Programmers make the internal state observable by inserting diagnostic print statements in the code or by using a debugger (e.g., GDB[1]), which enables the programmer to pause the program's execution and inspect the values of program variables.

Cyclic debugging is an effective means for verifying hypotheses. For example, consider Figure 2.1, which depicts a defective program for computing the mean and standard deviation for an array of numbers. This program will serve as running example throughout this section. The defect is that the variable `sum` is declared as an `int` rather than a `double`, which leads to a rounding error. If the programmer hypothesizes that the cause of the failure is that the value returned by the call to `read_array` is incorrect (i.e., not the size of the array as expected), he can easily verify this hypothesis by inserting a statement that prints the value of `n` on line 16. He can then reproduce a failing run of the program to see if the value reported for `n` matches the expected value. In this case, the hypothesis would be refuted, and he would update his working set of hypotheses accordingly.

Cyclic debugging is also an effective means for narrowing the space of plausible hy-

[1] `http://www.gnu.org/software/gdb/`

```
#include <cstdlib>
#include <cmath>
#include <iostream>

using namespace std;

int read_array(double a[]) { ... }

int main(int, char*[])
{
  double mean = 0.0;
  double stdev = 0.0;
  double a[BUFSIZ];
  int sum = 0.0;

  int n = read_array(a);

  for (int i = 0; i < n; ++i) {
    sum += a[i];
  }

  mean = sum/n;

  for (int i = 0; i < n; ++i) {
    a[i] = a[i] - mean;
    a[i] *= a[i];
  }

  sum = 0.0;

  for (int i = 0; i < n; ++i) {
    sum += a[i];
  }

  stdev = sqrt(sum/(n-1));

  cout << "mean:  " << mean  << endl;
  cout << "stdev: " << stdev << endl;

  return 0;
}
```

Figure 2.1: Defective program for computing mean and standard deviation.

potheses. For example, the ACE Toolkit[2] includes a class `ACE_Trace` that provides facilities for logging which class methods are entered and exited during an execution of the program [56]. The programmer can use the log produced during a failing run of the program to focus his attention on the methods that are most likely to contain the defect—that is, the methods that execute during a failing run. Focusing on the such methods frees the programmer from considering hypotheses involving code that never actually executes during a failing run.

Dependence Analysis

Dependence analysis involves determining how program statements may influence one another at runtime [21]. Programmers typically perform dependence analysis implicitly as they debug [130]. They are primarily concerned with two types of dependences: *data* and *control* dependences. A statement s is data dependent on a statement t if there is a variable v that s references and t defines (i.e., assigns to), and t may be the last statement to define v before s executes. For example, in the code:

```
x = 10;
cout << x << endl;
```

the print statement is data dependent on the assignment statement because the value of `x` to be printed will have been defined by the assignment. A statement s is control dependent on a conditional statement t (e.g., an if-statement or a while-loop) if one of t's branches must always lead to s and another branch may bypass s. For example, in the code:

```
if (x > 0) {
    ++x;
}
```

[2] http://www.cs.wustl.edu/~schmidt/ACE.html

the increment statement is control dependent on the if-statement because the conditional determines whether the increment will execute. A statement's data and control dependences directly influence the statement; however, the statement is also influenced, albeit indirectly, by its transitive dependences.

Dependence analysis can effectively reduce the space of plausible hypotheses during debugging. For example, based on a failing run of the program in Figure 2.1, the programmer can observe that the value of the variable `mean` printed by the statement on line 38 was incorrect. By following the transitive dependences backward from that statement, the programmer focuses his attention on the statements that could have influenced the execution of the print statement. These statements constitute a *slice* with respect to the print statement (Section 2.1.3). The programmer can ignore the statements that do not influence the print statement—that is, he can disregard any hypotheses he might have related to those statements. In this case, he can ignore the statements on lines 25–36—roughly half of the statements in the program.

Programmers often identify defects in code via *bug smells*—that is, the recognition that a section of code bears characteristics of a known type of defect [49]. They may notice suspicious-smelling code at any time during debugging. Programmers use dependence analysis to verify that a suspicious section of code could influence the observed failure. For example, suppose the programmer is trying to find the defect in Figure 2.1 that caused an incorrect mean to be displayed, and he detects a suspicious smell from the for-loop starting on line 25. He can check whether the for-loop could have affected the value of the variable `mean` when it is printed on line 38 by tracing the transitive dependences forward from the for-loop. In this case, he would find that the for-loop cannot influence the value of `mean` and therefore does not contain the defect.

2.1.2 Challenges of Concurrency

Concurrent programs may be more responsive or perform faster than sequential programs; however, they are more complex, and this complexity creates a number of problems during debugging. A sequential program is one whose instructions are executed sequentially by a single processor. In contrast, a concurrent program comprises a set of interacting sequential programs (or *processes*) that execute in abstract parallelism. The parallelism is *abstract* because there may only be one physical processor that the processes take turns executing on.

There are two main paradigms for expressing concurrency in programs: the message-passing and shared-memory paradigms. In the message-passing paradigm, processes have disjoint memory spaces and interact via the sending and receiving of messages. In the shared-memory paradigm, processes share a memory space and interact by writing and reading shared variables. Our work is concerned with multithreaded programs, which generally fall under the shared-memory paradigm. In multithreaded programs, processes are referred to as *threads*.

In multithreaded systems, memory accesses by threads are ordered—that is, two threads cannot access the same memory location at the same time. Two threads that are concurrently vying for access to the same shared variable gain access to the variable in an order determined by the *thread scheduler*, which decides the time slices that each thread gets to execute. As threads receive their time slices, their instructions *interleave*. Thread scheduling is nondeterministic, so for a given program, many different interleavings are possible. As a result, different runs of the program, which are given the same inputs, may yield different results. For example, Figure 2.2(a) depicts a trivial multithreaded program where functions `T1` and `T2` are executed concurrently by different threads. Even with this simple multithreaded program, many different resulting values for `x` and `y` are possible depending on the thread schedule (summarized in Figure 2.2(b)[3]). In this case, almost every schedule

[3] In actuality, the problem is worse than the figure suggests because interleaving may occur at a finer level

```
X: volatile int x = 0;
Y: volatile int y = 0;

   void T1()
   {
A:      x = 10;
B:      y = x + 10;
   }

   void T2()
   {
C:      x = y;
D:      y = 0;
   }
```

(a)

Schedule	x	y
ABCD	20	0
ACBD	0	0
ACDB	0	10
CABD	10	0
CADB	10	20
CDAB	10	20

(b)

Figure 2.2: Simple multithreaded program (a) and potential thread schedules and results (b).

produces a different result.

Nondeterministic scheduling creates the potential for a type concurrency-specific defect: *race conditions*—that is, the situation where certain interleavings result in error behavior and the correct behavior of the program depends on which thread schedule is used [87]. For example, Figure 2.3 depicts an implementation of a *producer-consumer program* [10] that exhibits race conditions. The producer-consumer program comprises threads that produce characters (lines 28–32) and threads that consume characters (lines 34–38). The characters are passed between the threads via a shared buffer (lines 1–25). In this example, threads do not synchronize their accesses to the buffer and may interfere with one another. Such interference may corrupt the buffer. For example, consider two consumers C_1 and C_2 concurrently calling `get` on a buffer with one item. C_1 starts executing in `get` first but is preempted by C_2 immediately after the while-loop. C_2 executes the entire call to `get`, leaving the buffer empty. Finally, C_1, having already passed the while-loop, decrements `size`, which results in the buffer having a negative size—a clear error.

of granularity than the program-statement level, depending on how the compiler translates the source code.

```
class Buffer
{
public:
  Buffer() : size(0) {}

  void put(char c)
  {
    while (size == BUFSIZ) /* do nothing */;

    carray[size] = c;
    ++size;
  }

  char get()
  {
    while (size == 0) /* do nothing */;

    --size;
    return carray[size];
  }

private:
  int size;
  char carray[BUFSIZ];
};
```

```
Buffer buf;

void T_producer()
{
  char c = produce_a_char();
  buf.put(c);
}

void T_consumer()
{
  char c = buf.get();
  consume_a_char(c);
}
```

Figure 2.3: Producer-consumer example with race conditions.

To avoid race conditions, threads must *synchronize* their accesses of shared variables—that is, they must collaborate to ensure that they do not interfere with one another as they access the variables. The most commonly-used synchronization mechanisms for multithreaded programs are *mutual-exclusion locks* (or *mutexes*), which can ensure threads have mutually exclusive access to shared variables, and *condition variables*, which enable threads to conditionally wait [18]. For example, Figure 2.4 depicts a new implementation of the buffer that uses locks and condition variables to prevent the race conditions in the previous implementation. The buffer is implemented using the *monitor* design pattern [50, 107]. Following the monitor pattern, the buffer has a mutex (or *monitor lock*) (line 39). Clients (i.e., producer and consumer threads) must acquire the lock when they enter a buffer method (lines 8 and 22) and release it when they return (lines 17 and 31). This ensures that only one client can execute within the methods of buffer at a time (i.e., the clients have mutually exclusive access to the buffer).

The buffer also uses condition variables (lines 40 and 41) to make producers and consumers wait while the buffer is full and empty, respectively. A producer that calls `put` while the buffer is empty will block within a call to `wait` on the `nonfull` condition variable (line 10). When a consumer removes a character from a full buffer, it calls `broadcast` on the `nonfull` condition variable (line 29), which unblocks any waiting producers. Consumers wait for a nonempty buffer in an analogous fashion.

Unfortunately, incorrect synchronization can lead to other types of concurrency-specific defects, such as *deadlock*, which occurs when two or more threads are mutually waiting on one another and none can make progress [10]. For example, Figure 2.5 depicts a variant of the producer-consumer program with multiple buffers (represented as a UML class diagram [102]). The producer and consumer threads interact with the buffers via a buffer manager (`Buffer_Manager` in the figure). If the buffers are implemented as in Figure 2.4 and the buffer manager is implemented as a monitor, then the program may deadlock. To understand why, consider a scenario where a consumer thread attempts to get a character

```
class Buffer
{
public:
  Buffer() : size(0), nonfull(lock), nonempty(lock) {}

  void put(char c)
  {
    lock.acquire();

    while (size == BUFSIZ) nonfull.wait();

    carray[size] = c;
    ++size;

    if (size == 1) nonempty.broadcast();

    lock.release();
  }

  char get()
  {
    lock.acquire();

    while (size == 0) nonempty.wait();

    --size;
    char rval = carray[size];

    if (size == BUFSIZ - 1) nonfull.broadcast();

    lock.release();
    return rval;
  }

private:
  unsigned size;
  char carray[BUFSIZ];

  Mutex lock;
  Condition nonfull;
  Condition nonempty;
};
```

Figure 2.4: Producer-consumer buffer implemented as a monitor.

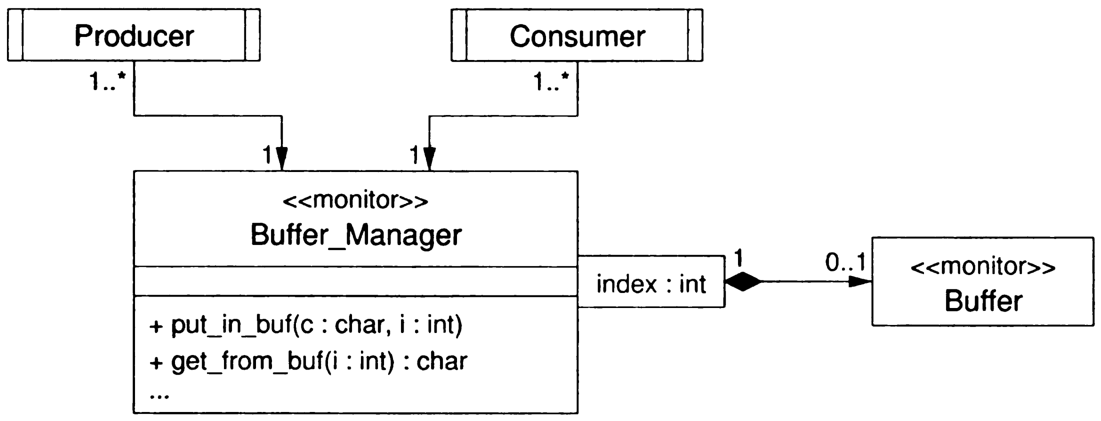


Figure 2.5: Deadlocking variant of the producer-consumer program with an array of buffers.

from the buffer at index i. If the buffer is empty, the consumer will wait on the buffer's `nonempty` condition variable, releasing the buffer's lock in the process. However, the consumer will still be holding the buffer manager's lock. No other clients of the buffer manager will be able to acquire the lock to access the buffers, thus the threads will deadlock.

Determining whether a program has concurrency-specific defects is generally difficult. Programmers may have an especially difficult time discovering defects that only manifest under certain subtle thread schedules. To determine whether a program is free from such defects, the programmer must consider all possible thread schedules. In the case of deadlock, the programmer must reason globally about such schedules. Unfortunately, the number of schedules grows exponentially with the number of threads, making it difficult to understand all the potential behaviors the program may have. Furthermore, testing may not reveal the defect because a failing schedule is not chosen.

Even when a programmer knows that a program contains a defect, concurrency makes diagnosing the defect difficult by rendering cyclic debugging and dependence analysis ineffective. Cyclic debugging is ineffective because it requires repeatedly producing failing runs of the program; however, failing runs may not be reproducible if the failure depends on subtle thread schedules. Moreover, adding print statements to the code may cause a

Table 2.1: Data dependences for each thread schedule of the program in Figure 2.2.

Schedules	Data Dependences
ABCD	$\{ A \xrightarrow{dd} B, B \xrightarrow{dd} C \}$
ACBD, ACDB	$\{ Y \xrightarrow{dd} C, C \xrightarrow{dd} B \}$
CABD, CADB	$\{ Y \xrightarrow{dd} C, A \xrightarrow{dd} B \}$
CDAB	$\{ Y \xrightarrow{dd} C, D \xrightarrow{dd} B \}$

probe effect where the presence of the statements affects how threads are scheduled and as a result, prevents the defect from manifesting [44].

Dependence analysis is ineffective because reasoning about data and control dependences in concurrent programs is difficult. In a sequential program, the program statements execute sequentially, which enables the programmer to trace the dependences through the code with relative ease. In a concurrent program, control flows nonsequentially through the code as the scheduler switches between different threads. This makes reasoning about inter-thread dependences difficult. Moreover, the data dependences vary depending on the thread schedule. For example, Table 2.1 lists the data dependences associated with each thread schedule for the program in Figure 2.2. In the table, $a \xrightarrow{dd} b$ denotes that statement b is data dependent on statement a.

2.1.3 Attempts to Address the Challenges

The literature includes numerous approaches that address the difficulty of debugging concurrent programs. *Replay* approaches address the problems with cyclic debugging by making executions deterministic. Automated *slicing* approaches perform dependence analysis to compute all the statements that a specified target statement is data or control dependent on. Still other approaches, such as those for detecting defects, provide information that is useful during debugging. In this section, we survey the work related to debugging concurrent programs.

Replay

Replay involves two steps: *recording* and *playback*. During recording, the program is executed and the replay system dynamically collects information, such as the order in which instructions were executed or the values of program data at different points. During the playback, the replay system uses the recorded information to faithfully reproduce the original execution. Some nondeterminism is acceptable during playback as long as the order of the interactions between threads is the same as in the original execution.

Programmers can apply a replay system to debugging in two ways. In the first use case, the programmer continuously records a deployed program—that is, a program being used "in the field." In the second use case, deployed programs are not recorded, and the programmer attempts to reproduce and record a failing execution "in the lab." Recording a failing run that occurs while the program is deployed has the benefit that the programmer is not forced to reproduce the failure, which may be expensive and may depend on luck. If the programmer expends the effort of finding a way to reliably reproduce a failure, then he has defeated the purpose of the replay system in first place. Recording deployed programs has the drawback that recording inevitably degrades the program's performance, which may be unacceptable to the program's users.

The primary challenge to implementing a practical replay system is recording. For either of the two previous use cases, recording should be minimally intrusive so as not to degrade performance by interrupting the program's natural execution or using too many system resources. Moreover, for the second use case, the recording system should avoid or at least minimize the possibility of probe effect; otherwise, it may be impossible to reproduce (and record) the failure.

The literature includes two types of approaches to implementing replay: *contents based* and *ordering based*. Contents-based approaches emphasize the replay of individual processes. These approaches record all the messages that each process receives and all the shared data values that each process reads. During playback, a single process executes and is fed the recorded data by the replay system rather than actually interacting with other threads. Curtis and Wittie [34] implemented one of the earliest replay systems, BugNet, which was contents based and for message-passing programs. Unfortunately, in practice, contents-based approaches do not scale well because the large volume of data they record generates unacceptable runtime overhead.

Ordering-based approaches address the problem of recording too much data by only recording the ordering of events and allowing the program to recompute all the other data during playback. Carver, Tai, and colleagues [19, 20, 119, 120] developed the first ordering-based replay system, which only records the order of explicit synchronization events (referred to as a *synchronization sequence*), such as the invocation of mutex operations and message sends and receives. They store the synchronization sequence using a global queue of process identifiers. During playback, each process may only execute a synchronization operation if its identifier is next in the queue; otherwise it waits. LeBlanc and Mellor-Crummey [66] developed another ordering-based replay system, Instant Replay, which models synchronization operations as accesses of shared variables. They represent the ordering of such accesses as a sequence of version numbers. During replay, each process may only execute a synchronization operation if the associated variable has the same version number as in the original execution; otherwise, the process waits. Both of these replay systems have the limitation that the programs they replay may not contain *data races*—that is, unsynchronized accesses of shared variables. Although these ordering-based approaches are more efficient than their contents-based predecessors, they are still not efficient enough to be practical—especially at the level of granularity required to handle data races.

Netzer and Miller [86, 85] developed a replay system capable of supporting programs that contain data races. Their approach is based on the observation that the ordering of only a subset of process interactions needs to be recorded to enable playback—specifically, the ordering of interactions that represent races. The system detects such races *on-the-fly*—that is, as the program executes—and decides which interactions to record accordingly.

Levrouw and colleagues [72], and Choi and Srinivasan [25] developed similar systems that further reduce the recording overhead in exchange for less efficient playback. Despite the reductions in recording overhead, these replay systems still do not scale well at the level of shared-memory accesses. One reason for the problem is that their performance depends on the ability to efficiently identify which variables are shared, which is nontrivial in modern programming languages. Furthermore, a conservative approach that treats all variables as shared is too inefficient to be practical.

Ronsse, De Bosschere, and colleagues [45, 101] developed a replay system based on Levrouw and colleagues' approach that works at the level of explicit synchronization operations. However, like the Carver and LeBlanc systems, replay may be nondeterministic if the program contains data races. To address this problem, Ronsse, De Bosschere, and colleagues [101] incorporated a *dynamic data-race detector* (see Section 2.1.3) into their system to automatically check for the presence of data races during playback.

Russinovich and Cogswell [103] tried to support the replay of programs that contain data races by recording a complete thread schedule. They capture the schedule from the operating-system thread scheduler by recording the value of an *instruction counter* [80] at the points of preemption during an execution. The instruction counter has a unique value for each instruction in an execution of the program. During playback, the scheduler uses the stored counter values to decide when to preempt a process. Although their approach is highly efficient, it is limited to uniprocessor systems and depends on access to the thread scheduler that is not available under most commercial operating systems.

A final group of approaches attempts to address the inefficiency of recording with hardware support [6, 84, 136]. Although, these approaches achieve the best performance, they rely on experimental hardware architectures that are not generally available.

A general problem with cyclic debugging (and, by extension, replay) is that the failing execution may be overly long. The literature describes approaches for shortening such executions for sequential programs, such as through *delta debugging*—a general approach for reasoning about the differences between runs of a program [138]. However, these approaches are not effective for concurrent programs. Moreover, concurrent programs exacerbate the problem because a program containing a subtle concurrency defect may run for a very long time before the defect manifests. The work on *reversible execution* [95, 88] addresses this problem for concurrent programs by enabling the programmer to jump to intermediate points in the execution during playback. These approaches generally involve recording *checkpoints*—that is, snapshots of the global system state—during the original execution. Unfortunately, checkpoints tend to be very large making these approaches too inefficient to be used in practice.

Slicing

Slicing is an automatable application of dependence analysis [121, 129, 135]. It is a program-transformation technique that, given a *slicing criterion*, removes statements in the program so that the remaining statements represent all the transitive data and control dependences with respect to the slicing criterion. A slicing criterion is typically a statement or a subset of the variables used in a statement. The remaining statements (or *slice*) may constitute an executable program. Automated slicing was originally proposed as a way to reduce debugging effort [129], but has since found a number of other applications, including program comprehension [75], testing [97], and formal verification [48].

Automating slicing is technically challenging and has been a hot area of research (cf. surveys by Tip [121], and Xu and colleagues [135]). Automated slicing solutions attempt to address several key issues: (1) *slice precision*, (2) *slice recall*, and (3) efficiency of slicing algorithm. Slice precision decreases as more statements that cannot influence the slicing criterion are included in the slice (i.e., a slice with no statements is 100% precise by default). Slice recall increases as more statements that can influence the slicing criterion are included in the slice (i.e., a slice with all the statements in the program has 100% recall by default). Modern programming language features, such as procedures, pointers, dynamic binding, and exception handling, make it difficult to compute slices with both high precision and high recall. Solutions tend to be *conservative*—that is, they sacrifice precision to ensure recall. This makes sense for debugging where a slice that may not contain the defect is of little value. Furthermore, the computations required to produce precise slices are often inefficient or even intractable. Solutions attempt to strike a balance between efficiency and precision—often sacrificing precision for a simpler, more-efficient slicing algorithm.

Automated slicing solutions are most commonly based on a graphical approach [93]. First, the slicer generates a *control-flow graph* (CFG) for the program, which is a graphical representation of all the paths that might be traversed through the statements of a program during execution [1]. For example, Figure 2.6 depicts the CFG for the program from Figure 2.1. Second, the slicer transforms the CFG into a *program dependence graph* (PDG), which has the same nodes as the CFG, but the nodes are connected by arcs representing their control and data dependences. For example, Figure 2.7 depicts the PDG for the program from Figure 2.1. Finally, given a slicing criterion (e.g., a statement in the program), the slicer computes a slice by computing all the nodes that are reachable from the slicing criterion by traversing dependence arcs. For example, Figure 2.8 depicts the nodes reachable from the statement on line 37 (i.e., the slice on line 37).

The previous example depicted *static slicing*, which computes a program's dependences by analyzing the source code. An alternative approach is *dynamic slicing*, which computes dependences that appear during a particular execution of the program—that is, statements that did not execute during the run will be omitted from the slice [61]. Dynamic slicing has the advantage of producing smaller slices, which are tuned for particular executions. In debugging, the smaller slice creates a smaller search space for the defect. However, the slice must be based on a failing execution, which may be difficult to produce for concurrent programs.

The original work on slicing concurrent programs took a dynamic approach and focused on message-passing systems. Korel and Ferguson [60] log the sequence of statements (or

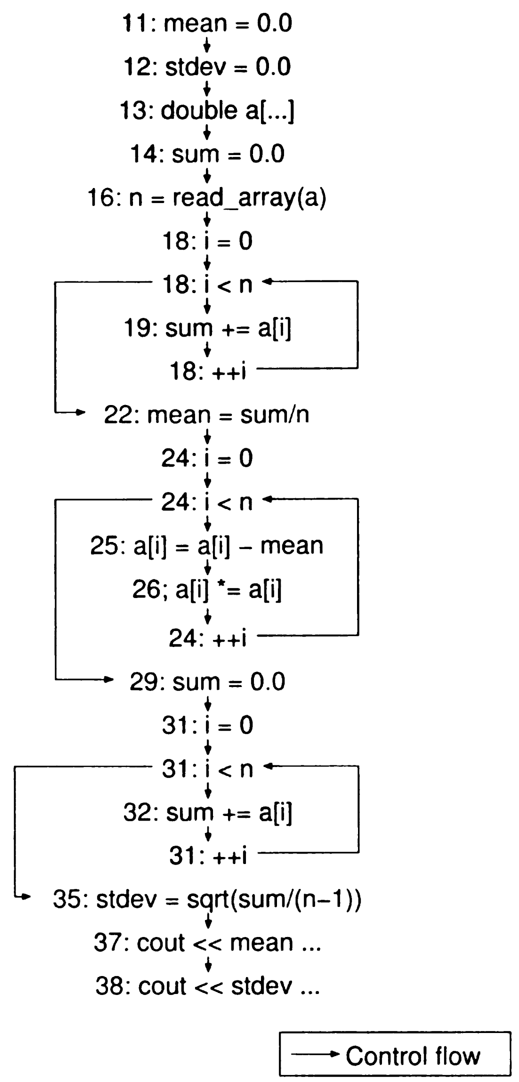


Figure 2.6: Control-flow graph (CFG) for the program in Figure 2.1.

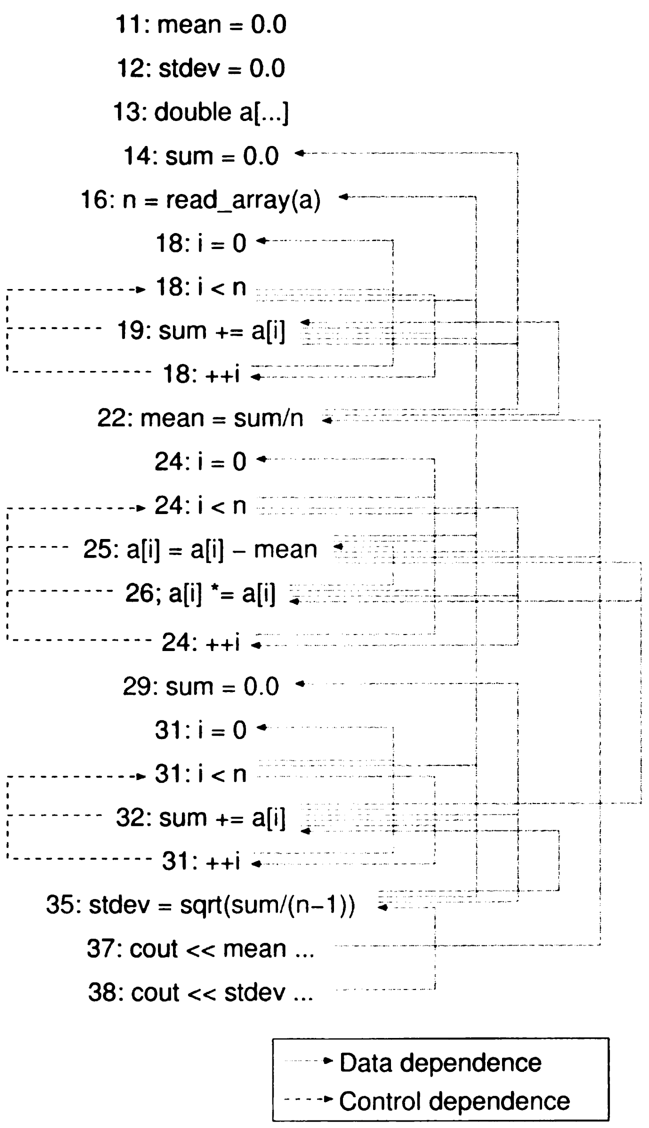


Figure 2.7: Program-dependence graph (PDG) for the program in Figure 2.1.

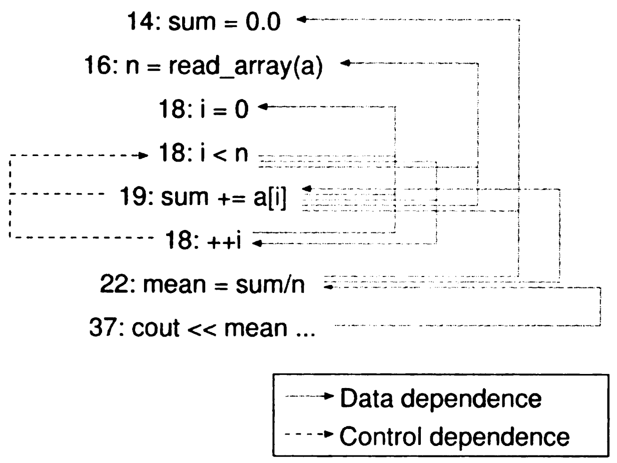


Figure 2.8: Slice on line 37 of the program in Figure 2.7.

trace) that each process executes during a particular run of a program. They compute intra-process data and control dependences for each trace. To capture inter-process data and control dependences, they associate each message-send operation with its associated receive operation—called *communication dependence*. Computing the slice involves computing all the statements that are reachable from the slicing criterion by following dependences. Duesterwald and colleagues [35] statically compute control dependences in the form of a *control dependence graph* and insert data and communication dependences into that graph on-the-fly. Their approach avoids the overhead of storing trace data. However, they sacrifice precision in the presence of loops because a single node is used for all executions of a statement in the loop. Kamkar and Krajina [57] extended Korel and Ferguson's approach with support for procedure calls. None of the previous dynamic slicing approaches address the problem of producing a failing run of the program.

Cheng [23] developed a static-slicing approach for message-passing systems by extending the previously described graphical slicing approach. In Cheng's approach, each process essentially has its own CFG. Intra-process data and control dependences are computed as before. He represents inter-process data dependences as *communication dependences* and inter-process control dependences as *synchronization dependences*. A statement s is communication dependent on a statement t if s is data dependent on a message-receive statement and an associated message-send statement is data dependent on t. A statement s is synchronization dependent on a statement t if the start or termination of execution of t determines whether s starts or terminates—for example, in the case where t is a message-send statement and s is an associated message-receive statement. Zhao and colleagues [141] extended Cheng's approach with support for object-oriented language features. Zhao [140] further extended it with support for multithreaded Java features, such as condition synchronization; however, he did not address the dependences that arise from unsynchronized accesses to shared variables. The slices computed by these approaches are imprecise because they fail to take into account that inter-thread data dependences are not transitive.

Krinke [62] first addressed the static slicing of shared-memory based concurrent programs. Like Cheng, he also uses the graphical approach, except that he does not compute the slice using simple graph reachability. Rather, his slicing algorithm uses symbolic execution to address the intransitivity of inter-thread data dependences. To capture inter-thread data dependence, he introduces *interference dependence*. A statement s executed by thread T_s is interference dependent on a statement t executed by thread T_t ($T_s \neq T_t$) if there exists a schedule such that t defines a variable v, which is referenced in s, and s's execution follows t's with no intervening definition of v. Krinke's original approach was in the context of an extremely simple language, which lacked many common features, such as loops and synchronization mechanisms. Nanda and Ramesh [83] extended Krinke's approach with support for nested threads and loops. Krinke [63] extended his original work with support for procedure calls. This support takes calling context into account, which produces more precise slices. Hatcliff and colleagues [48] extended Krinke's approach for multithreaded Java. In particular, they added support for Java's monitor synchronization mechanisms. To represent the inter-thread control dependences these mechanisms create, they introduced *ready dependence*. A statement s executed by thread T_s is ready dependent on a statement t executed by thread T_t (T_s may equal T_t) if there exists a schedule such that s will not complete unless t does. Ranganath and Hatcliff [99] addressed the inefficiency of Krinke's symbolic-execution based slicing algorithm with an approach based on escape analysis; however, their approach sacrifices some precision.

Other Approaches

Numerous other tools and methods provide support for the debugging of concurrent software. Traditional parallel debuggers behave like a collection of sequential debuggers (e.g., GDB) such that processes (and their console output) are displayed individually [77]. Like sequential debuggers, they allow the programmer to pause the program's execution by setting breakpoints and to inspect various aspects of the program's state. The programmer can perform such operations on individual processes or on groups of processes. The utility of parallel debuggers is rather limited in that they do not address probe effect, do not address the difficulty of reproducing failures, and display information at the level of program instructions, making it difficult to reason about what is happening at the level of process interactions.

Similar to replay, event-history browsers [30, 67, 98, 114] record events during a program's execution and allow the user to subsequently inspect the recorded events. They typically store the events in databases, and inspection involves querying the databases. This type of approach does not scale well as it imposes high overhead during recording.

The literature includes a number of approaches for detecting the presence of errors in concurrent programs. These approaches generally provide feedback when they detect an error that can aid debugging. Race detectors automatically discover data-race errors in concurrent programs. There are two main approaches to race detection: static and dynamic. Static race detectors [2, 41] discover data races by analyzing the program sources. They tend to be imprecise, reporting a large number of spurious races, because modern programming language features, such as pointers, make predicting what data is shared difficult or undecidable. Dynamic race detectors discover data races by analyzing the events during an execution of the program. They perform the analysis on a recorded event history [81] or on-the-fly [22, 24, 91, 106, 128]. They tend to be more precise than static detectors because they analyze the events that actually occur during an execution. However, they tend to have reduced recall, missing some races, because parts of the program that do not execute during the run will not be analyzed. When a race detector detects a race, it provides feedback to aid debugging, such as the statements and variables involved in the race.

Model checking [17, 26, 27, 28, 31, 36, 55, 76] detects errors by automatically analyzing whether a program violates specified correctness properties. The approach involves constructing a behavioral model of the program. If the model checker finds that a property is violated, it reports a counterexample describing the behavior that leads to the violation. In practice, models that represent all possible program behaviors cannot be analyzed because they are intractably large. To make analysis tractable, models must be sufficiently abstract, omitting aspects of the program that are not salient to the analysis. However, the more abstract the model, the less it resembles the program's implementation. Moreover, the modeling languages accepted by model checkers use very different constructs than the programming languages. These differences make it difficult to trace features in the model to features in the code. Thus, it is difficult to tell if the model accurately represents all the salient features of the program. Moreover, it is difficult to relate the behavior described in a counterexample to the associated part's of the program's implementation.

Formal proof techniques [4, 94] provide another way to determine whether a program satisfies correctness properties. They offer very little automated support, and the programmer typically must do much of the proof by hand. In performing the proof, the programmer makes observations about where in the code violations of properties occur—for example, he may discover that parts of the program violate a specified invariant. Such observations are useful for debugging. Unfortunately, proof techniques are difficult to apply correctly, and programmers rarely use them.

2.2 Empirical Methods

Selecting appropriate empirical research methods is key to achieving useful and valid results. A method must be capable of addressing the research questions of interest. Equally important, the method must produce answers to the questions that the target research community will accept as scientific truth. However, research communities, and even people within those communities, vary in their assumptions about scientific truth. This variation arises from differences in *philosophical stance*. The predominant philosophical stance in software engineering is *positivism* [92].[4]

Positivism states that "all knowledge must be based on logical inference from a set of basic observable facts" [37]. Positivists believe that scientific knowledge is built up incrementally from verifiable observations. Positivism emphasizes the use of *quantitative methods*, which are concerned with quantifying relationships or comparing two groups to identify cause-effect relationships [131]. Phenomena tend to be studied in highly controlled laboratory environments—that is, in isolation from their natural context. Unfortunately, software engineering phenomena tend to involve large-scale and highly-complex contexts and have important aspects that are difficult to quantify. This situation makes it difficult to validate that strictly positivist findings will generalize to practice.

In recent decades, software-engineering researchers have begun using *qualitative methods* to address the shortcomings of the strictly positivist approach. These methods are concerned with studying phenomena in their natural contexts. Qualitative data emphasizes words and pictures, not numbers. Qualitative methods address the full complexity of a phenomena rather than abstracting complexity away like quantitative methods do. They are useful for answering "why" questions that involve variables that are difficult to quantify. Qualitative results have the drawback that they tend to be difficult to simplify and summarize, and are often regarded as "softer" than quantitative results. Using *mixed meth-*

[4] Other philosophical stances include *constructivism*, *critical theory*, and *pragmatism*, and are beyond the scope of this dissertation.

ods—that is, combinations of quantitative and qualitative methods—can compensate for the weaknesses of some methods with the strengths of others. Our work incorporates both qualitative and quantitative methods.

2.2.1 Qualitative Methods

The question of what strategies successful programmers use to debug concurrent software is exploratory in nature—that is, it is concerned with generating a new theory. We base our approach to theory generation on the *grounded theory* approach [32]. It emphasizes the collection of rich qualitative data. Through iterative analysis, we develop a theory that fits the data. In the current work, we employ several qualitative methods for data collection and analysis.

We base our data collection on *think-aloud observation* [39, 109, 122]. This method involves observing participants as they engage in some activity. Participants "think aloud" as they perform the activity—that is, they verbalize their internal monologue (or stream of consciousness). The method provides insight into participants' goals and intentions—information that is particularly useful for identifying strategies. The data collected must be highly detailed to facilitate fine-grain analysis. We collect audio and video data of each performance to ensure that important details are not lost.

Coding constitutes the foundation of analysis. It involves the identification of *concepts*, which are classes of objects, events, and actions that occur in the data [32]. Each concept is assigned a unique identifier, which is referred to as a *code*. The researcher also develops a *coding scheme*, which specifies how the concepts can be identified in the data [122]. For example, a researcher might associate a code guessing with each of a participant's utterances that are of the form "Maybe it is *X*" or "Let's try *X*." The codes can be treated as quantitative variables, which can be counted or subjected to other statistics [109]. Because coding may be subjective, the validity and consistency of codes is a major concern. To address this concern, multiple researchers code the same data and compare their results (referred to as a *rater-agreement exercise* [109]). The researchers must reconcile any disagreement, refining their coding scheme in the process.

The *constant comparison* method prescribes a process for iteratively creating and applying codes, and developing explanations of phenomena [32]. Initially, data is collected and repeatedly reviewed and re-reviewed to formulate a hierarchy of codes. At this stage, tentative explanations for the underlying phenomena are formed. In subsequent rounds of data collection and analysis, the codes and explanations are reevaluated and refined as necessary. Data may be reanalyzed during the refinement stage as well.

The *cross-case analysis* method divides the data into cases to be compared [109]. Using different divisions, the data can be analyzed in different ways. For example, we are concerned with comparing the strategies of successful programmers with unsuccessful. Using cross-case analysis, we divide our data into the sessions of successful and unsuccessful programmers. We then compare the data of the successful and unsuccessful cases to see if there are differences in the strategies they used. We might subsequently divide the data into cases based on some other attribute to gain another perspective on the data. Back-and-forth analysis of cases is common in grounded theory research.

2.2.2 Quantitative Methods

Quantitative research emphasizes the numerical measurement of phenomena and the identification of cause-effect relationships [131]. In software engineering, the three most common strategies for quantitative research are *surveys*, *case studies*, and *controlled experiments* [40]. Surveys are retrospective studies of events that involve investigating relationships and outcomes after the fact. In contrast, case studies and controlled experiments are not retrospective—that is, they involve making observations as events actually occur. Case studies investigate a phenomenon within its real-life context, whereas controlled experiments are conducted in a carefully regulated environment, such as a laboratory. Controlled experiments are most closely associated with quantitative research [131]. We conducted an experiment to test a hypothesis that emerged from our qualitative research.

Experiments are designed to investigate causal connections between measurable phenomena [131]. Our work is concerned with the investigation of phenomena associated with the performance of tasks by human participants. The experimenter manipulates some factor of the experimental task to measure the effect. The manipulated factor and the measured effect are referred to as the *independent variable* and *dependent variable*, respectively. If the experiment is successful, then the effect on the dependent variable depends on the manipulation of the independent variable. Thus, the experimenter can make inferences about a causal relationship between the variables. A challenge in experimental studies is keeping all other factors constant, so a clear comparison of the effects on the dependent variable can be made. A *confounding variable* is a factor other than the independent variable that affects the dependent variable and that may mask the effects of the independent variable.

2.3 Related Empirical Studies

In this section, we describe the empirical studies from the literature most closely related to our own work. These studies predominantly investigated the detailed behaviors of programmers related to debugging. We also include studies that focused on program comprehension, an activity closely related to debugging. The majority of this work is exploratory in nature and involves the collection and analysis of qualitative data (esp. observations of programmers at work).

Gould and Drongowski [46, 47] conducted early studies of programmers engaged in debugging. In particular, they observed students and professionals diagnosing defects in small (less than 100 SLOC) FORTRAN programs. The studies took place in a controlled laboratory environment. Both quantitative and qualitative data yielded several interesting results. Gould and Drongowski found that programmers "ease into" debugging—that is, they put off dealing with more complex parts of the code until all other options are exhausted [47]. They found evidence that simply reading code yields defect diagnoses faster than cyclic debugging [47]. Through qualitative analysis, Gould derived a high-level model of the debugging process, which emphasizes an iterative process of generating, verifying, and refining hypotheses regarding the defect [46].

Weiser [130] investigated whether programmers implicitly perform slicing when they debug. Specifically, he wanted to see if programmers mentally construct slices by tracing data and control dependences backward through the code to find the source of an error. He conducted a controlled experiment in which programmers debugged a program and were tested to see if they remembered relevant slices as well as relevant contiguous code. He found that programmers indeed remember slices, and concluded that they implicitly use a slicing strategy when they debug.

Vessey [123] also studied the behaviors of programmers engaged in debugging. She used the think-aloud method to investigate the debugging practices of expert and novice programmers. The results suggest that experts tend to solve a debugging problem using a deliberate and precautionary strategy of hypothesis generation and validation. This *breadth-first problem-solving 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, a *depth-first approach* involves attempting to verify hypotheses as they are formed—prior to gaining a high-level understanding of the problem. Furthermore, the results suggest that expert programmers use the breadth-first approach in conjunction with *system thinking*, which involves creating an implicit mental model of the program's structure and function, and that novice programmers use both breadth-first without system thinking and depth-first approaches.

Hochstein, and colleagues [9, 51, 52, 53, 54] constitute one of the few research groups to empirically study concurrent programmers. They studied the debugging habits of computational scientists who develop large concurrent applications [52]. However, scientific applications represent a fundamentally different type of concurrent software than the reactive server applications emphasized in our work. As such, there are marked differences in how the two types of applications are developed. For instance, scientific applications are commonly written using the MPI programming model [113], whereas reactive applications commonly use the POSIX Threads model [18]. The researchers collected data by interviewing scientists, which is in contrast to the detailed observations of programmers used in our work. They found that computational scientists depend heavily on cyclic debugging, which is performed using traditional parallel debuggers (e.g., TotalView[5]) and diagnostic print statements. However, they did not find any special techniques for replicating failures. They also noted that batch scheduling, which is commonly used with scientific applications, makes cyclic debugging more difficult because jobs can take a long time (as much as a week) to be scheduled.

Littman and colleagues [73] investigated the nature of program understanding by analyzing the strategies and knowledge programmers employ in performing perfective maintenance. The participants, who were professional programmers, added a feature to a small program, and a researcher asked them questions about their thoughts as they worked. The results from the study suggest that a *systematic comprehension strategy* is more effective than an *as-needed strategy*. The systematic strategy involves starting at the beginning of the program and tracing the flow of the entire program, using various forms of simulation. In contrast, the as-needed strategy involves studying only those portions of the code that are believed to be useful for the task at hand. Furthermore, the results suggest that two distinct kinds of knowledge—*static* and *causal*—must be gained during systematic comprehension. Static knowledge refers to an understanding of a program's functional components (e.g., roles, classes and methods), whereas causal knowledge refers to an understanding of how the functional components interact at run time.

Several recent studies have called into question the usability of the systematic comprehension strategy for large programs [59, 124, 100]. These studies involved observing

[5] http://www.totalviewtech.com/

participants performing maintenance tasks on programs that were significantly larger than that used in the original study. The results of the studies suggest that no participant used the systematic strategy because of the limited scalability of the strategy. In addition, the results of one of the studies suggest that in lieu of the systematic strategy, a *methodical approach* to change tasks leads to success [100]. Such an approach involves investigating enough of the code to understand the high-level structures of the system, preparing a plan of the change to be made, and implementing the plan in a linear fashion. In contrast, an approach that is ad hoc and opportunistic tended to lead to failure.

Von Mayrhauser and Vans [124] investigated the information needs of programmers comprehending programs in the context of debugging. Their work is notable because it is among the first to study programmers working on large-scale software (greater than 40,000 SLOC). They found that programmers who are engaged in debugging familiar programs emphasize the need for low-level data and control flow knowledge. This low-level knowledge is in contrast to higher-level knowledge about implementation plans related to the application domain. They also found that managing hypotheses about code tends to be taxing on cognitive resources, such as working memory. To address the problem, they recommend the development of a tool to help manage such hypotheses.

Two related studies investigated the process programmers use for gathering and using information during maintenance [58, 65]. These studies were both geared toward informing the design of integrated development environments (IDEs). In the first study, participants performed several corrective and perfective maintenance tasks on a small program [58]. The results of this study suggest that programmers use a process of searching for, relating, and collecting information wherein the ability to evaluate the relevance of information is key. Based on these results, the authors recommend that IDEs should provide clear cues to help programmers judge the relevance of information, and support for collecting the information the developer deems relevant. The second study used a think-aloud design and involved participants performing refactoring tasks on a large (roughly 55,000 LoC) program [65]. The results of this study suggest that program comprehension is driven by the seeking, evaluating, explaining, and relating of facts about the program's design and implementation. Moreover, a key factor in the comprehension process was the programmer's uncertainty that he had correctly grasped all the relevant facts. Based on the results, the authors suggest that a tool that explicitly represents facts and maps them to parts of the code could help developers work more effectively. Such a tool was previously developed to address the *concept-assignment problem*, which is the problem of discovering human-oriented concepts, such as computational intent from the application domain, and relating those concepts to implementation structures in the code [12].

Several think-aloud studies investigate the questions programmers ask during maintenance tasks [69, 111]. A goal of these studies is to develop a taxonomy of questions, which may be used to inform the design of tools and documentation. In the first study, participants added a feature to a small program [69]. The results of this study suggest that programmers ask five types of questions: (1) *why* questions regarding the role pieces of code play, (2) *how* questions regarding the method for accomplishing a goal, (3) *what* questions regarding what a variable or function is, (4) *whether* questions regarding whether code behaves in a certain way, and (5) *discrepancy* questions regarding confusion over a perceived inconsistency. Two subsequent studies (conducted in tandem) extend the findings of the first study by incorporating larger, more-realistic programs and seeking to develop a richer, more comprehensive taxonomy of questions [111, 112]. The results of these two studies yielded a catalog of 44 types of questions organized in four categories related to the level of program understanding reflected in the questions.

Numerous studies have driven the development and validation of theories regarding the cognitive processes and knowledge structures programmers use during comprehension and maintenance tasks (e.g., [69, 96, 115, 117, 125, 126]). These studies generally employ the think-aloud method. Unfortunately, the theories tend not to be rendered at the level of detail necessary to address the issues of concurrency.

2.4 Notations and Visualizations for Thread Interactions

In this section, we describe some notations and visualizations that serve to externalize representations of thread interactions. First, we describe the UML sequence diagram notation, which is a focus of our work. Second, we describe our multithreaded extension to the sequence diagram notation. Lastly, we describe some other notations and visualizations of multithreaded-program behavior from the literature.

2.4.1 UML Sequence Diagrams

The Unified Modeling Language provides two notations for representing interactions: sequence diagrams and communication diagrams [102].[6] Sequence diagrams represent scenarios of execution by displaying the sequence of messages exchanged between objects. Figure 2.9 depicts a sequence diagram with various parts of the diagram labeled. The notation defines two kinds of objects: *active* and *passive*. An active object has an associated thread of control, whereas a passive object does not. Time advances down a sequence diagram. The objects are depicted as rectangles along the top of the diagram. The name and type of the object is given inside the rectangle. Bars on the left and right sides of the rectangle distinguish active from passive objects. A *lifeline* extends downward from each object. *Execution specifications* (or *activations*) represent the execution of methods and are depicted as bars that overlap the lifelines. An activation bar always covers the lifeline of an active object because the active object continuously executes a method, which contains a control loop. An object's lifeline may be adorned with *object states* to denote a change in the state of the object. Object states appear as roundtangles that contain a description of the new state (typically in the form of a predicate). Sequence diagrams represent the interactions between objects as a sequence of message sending actions between objects. A solid arrow labeled with the operation name and parameters denotes the call of an operation. A

[6] In this dissertation, we emphasize the use sequence diagrams to represent interactions and do not cover communication diagrams.

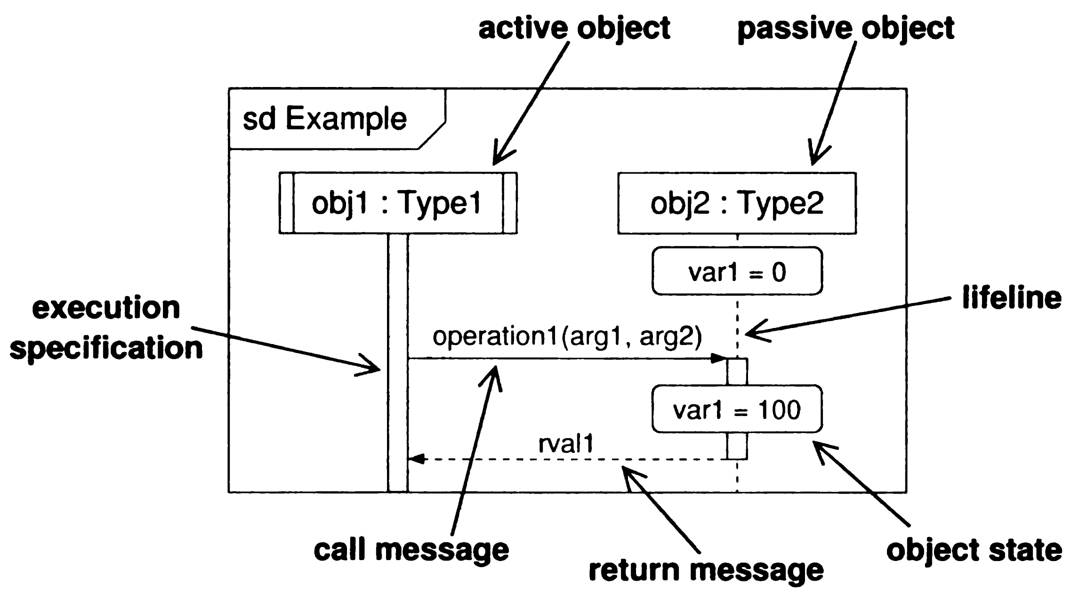


Figure 2.9: Example of a UML sequence diagram.

dashed arrow denotes the return from an operation and may be optionally labeled with a return value.

Figure 2.9 depicts an interaction between an active object obj_1 and a passive object obj_2. Initially, obj_2's attribute var_1 is zero. obj_1 calls the operation $operation_1$ on obj_2. The arguments to the call are arg_1 and arg_2. During the execution of the operation, var_1 is set to one hundred. Upon the completion of the operation, obj_2 returns the value $rval_1$ to the caller.

Sequence diagrams provide features that are useful for modeling interactions in multi-threaded systems. Threads can be modeled as active objects and shared resources as passive objects. The diagram visualizes the dynamics of synchronization, showing the ordering of interactions between threads in a single program trace. Unfortunately, the sequence diagram notation does not define a way to denote concurrent activations. Such activations commonly occur in thread interactions. Moreover, the sequence diagram notation lacks convenient features for expressing properties of multithreaded systems, such as a thread's *execution state*, which may be ready, running, blocking, or terminated. To address these

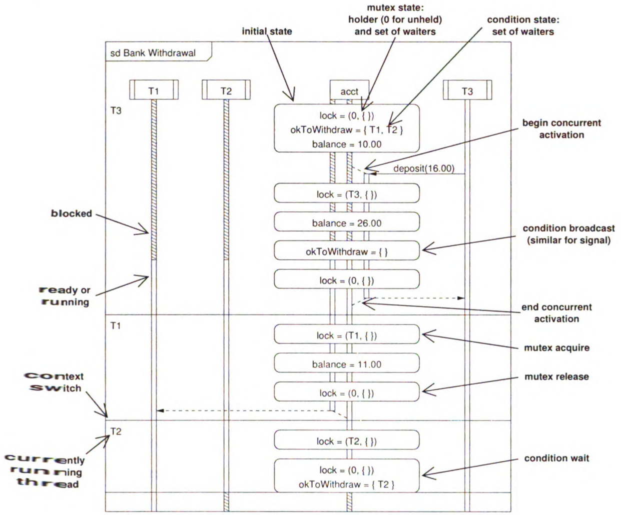


Figure 2.10: Multithreaded extentions to UML sequence diagram notation.

shortcomings, others have proposed limited extensions, which we describe in Section 2.4.3.

2.4.2 Multithreaded Extension to UML Sequence Diagrams

To address the lack of support for multithreading in standard UML, we designed a multithreaded extension to the sequence diagram notation. Figure 2.10 depicts an example that uses our extension with the features of the extension labeled. In the example, threads T_1 and T_2 are initially blocking within invocations of the *acct* operation *withdraw*.

Our extension uses several new notations to represent the execution state of each thread. A hatched activation bar indicates that the thread associated with the activation is in the blocked state. A non-hatched activation bar indicates that the associated thread is in the ready or running state. Horizontal lines that crosscut the entire diagram indicate context switches. Labels along the left hand side of the diagram indicate when the various threads run. The context-switch lines denote the point that the running thread changes.

The UML sequence diagram notation does not define how to denote concurrent activations of objects. We use branches of an object's lifeline to denote concurrent activations. When a concurrent activation ends, the branch merges once again with the original lifeline.

The UML sequence diagram notation does not provide a convenient way to denote mutex and condition variable state. We use such object states to denote the effects of invoking operations on mutexes and condition variables. We represent the state of a mutex as the pair (h, W). Here, h represents the holder of the mutex (0 indicates that the mutex is not held), and W represents the set of threads waiting on the mutex. In Figure 2.10, the first object state after T_3 calls *deposit* indicates that T_3 has acquired the mutex name *lock*. The object state immediately before T_3 returns from *deposit* indicates that T_3 has released the lock.

We represent the state of a condition variable as the set of waiters W. The bottommost object state in the figure indicates that T_2 waits on the condition variable *okToWithdraw*. This object state also depicts the release of *lock* by T_2. The T_2 activation bars become hatched immediately following the state roundtangle to indicate that T_2 begins blocking. Furthermore, a context switch occurs as a result of T_2 entering the blocking state.

2.4.3 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 animated images. Our work focues on UML sequence diagrams, which are static visualizations.

Others have extended sequence diagrams to better support concurrent software. Mehner and Wagner [78, 79] added shading conventions on execution specifications to indicate when and within which activation threads are ready or running. This convention implicitly denotes when a thread is blocking (all the thread's activations are unshaded); however, our convention makes blocking more explicit. Moreover, their extension also does not completely capture thread-state 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, which is invoked to lock an object. They model calls to Java synchronized methods using execution specifications that begin with a call of the form `synchronize(this)`. Likewise, entry into a Java synchronized block on some object o is modeled by a call of the form `synchronize(o)`. Because calls to this distinguished operation may block, this extension depicts the blocking of threads, which makes deadlocks easier to recognize. However, Mehner and Wagner's extension abstracts away subtle details, such as when locks are released during condition synchronization. Our extension represents this information using object states.

Xie and colleagues [132, 134, 133] also developed a concurrent extension to sequence diagrams. Their extension uses colored activations to indicate the state of each thread (i.e., running, blocking, or ready). With this approach context switches are implicit and not as apparent as with our notation. Moreover, their use of color makes the diagram inconvenient to draw by hand with a pen or pencil. It also precludes the use of colors to distinguish the activations of different threads—a feature that our extension can support. Xie and colleagues' extension assumes that all passive objects are monitors. This assumption reduces flexibility because, unlike our extension, they do not explicitly represent mutexes and condition variables. Like our extension, theirs uses object states to indicate when a monitor lock is held; however, unlike our extension, theirs does not explicitly represent the threads waiting on locks and condition variables.

Newman and colleagues [90] propose two new diagramming notations to statically visualize concurrency-related design decisions that are not easily derived through code inspection. Their *regional state hierarchy diagram* extends the UML class diagram to depict the structure of lock-state associations. The notation identifies the shared state within a set of classes and describes how that state is protected. Their *method concurrency diagram* extends a call graph to show which methods invoke operations on which mutexes and condition variables, which mutex protects each condition variable, and which data each lock protects. Although their diagrams do not model thread interactions explicitly, they provide supporting information that could be useful in conjunction with, for instance, a sequence diagram.

Traditional parallel debuggers (e.g., [77]) provide a rudimentary dynamic visualization by displaying a debugger window for each thread. Other dynamic visualization tools show the status of various properties as a multithreaded program executes. Leroux and colleagues [68] 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.

2.5 Thread-Interaction Complexity Metrics

As part of our controlled experiment, we needed to assess the complexity of thread interactions so that we could see whether complexity impacts the benefit of using external representations. We believe that the complexity of a thread interaction is driven not only by its size, but by a number of concurrency-related properties. For instance, it stands to reason that interactions involving more threads tend to be more complex than those involving fewer threads. Complexity also seems to increase with the level of contention. For example, an interaction over a monitor where context switches never occur while a thread is in the monitor (and thus no blocking occurs) seems less complex than a similar interaction where threads must block and unblock. Similarly, interactions with more context switches between threads may tend to be more complex than those with fewer context switches.

We have identified three properties that we suspect contribute significantly 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 based on labeled transition system (LTS) models of multithreaded programs.

2.5.1 Modeling Multithreaded Programs as LTSs

LTSs have been widely used to model concurrent programs [76]. In such models, states in the LTS represent abstract program states, and actions in the LTS represent atomic program instructions. For example, Figure 2.11 depicts an LTS model *Example* of a multithreaded program. The program comprises two threads t_1 and t_2, and each thread acquires and releases a shared lock infinitely often. Given an LTS model of a concurrent program, a thread interaction can be modeled as a *trace* of the LTS. A trace refers to the sequence of states and actions produced by executing the model. For example, Figure 2.12 depicts a trace *ExampleTrace* over the *Example* model.

We define our metrics over traces of an LTS, where we assume the LTS faithfully models the program and also possesses the properties described below. Others (e.g., [76]) have developed methods for modeling a program as an LTS. Our metrics assume that the model satisfies two properties. First, each transition in the model is associated with one and only one thread. Second, blocking and unblocking actions by threads are modeled explicitly. For instance, the *Example* model possesses these properties. Each action models the execution of exactly one thread. For instance, $t_1.acquire$ models thread t_1 acquiring the lock, whereas $t_2.acquire$ models t_2 acquiring the lock. Also, the actions $t_1.tryAcquireBlock$ and $t_2.releaseUnblockT1$ explicitly model t_1 entering and exiting the blocking state, re-

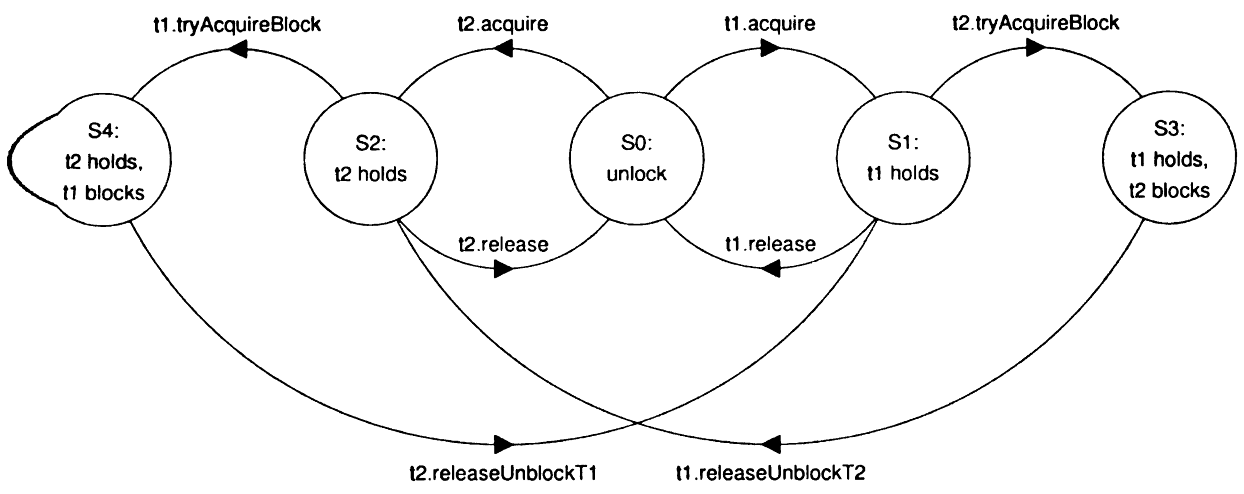


Figure 2.11: LTS model *Example* of a multithreaded program that comprises two threads t_1 and t_2 that each infinitely acquires and releases a shared lock.

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

Figure 2.12: One possible trace *ExampleTrace* of the LTS *Example* from Figure 2.11.

spectively.[7] The *Example* model has similar actions for blocking/unblocking t_2.

2.5.2 Formal Definitions

In this section, we formally describe a *multithreaded LTS*—that is, an LTS suitable for modeling a multithreaded program—and define the concept of traces over multithreaded LTSs. Based on these definitions, we define each of our metrics.

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 the 6-tuple $\langle S, T, A, \Theta, \Delta, q \rangle$ where $S \subseteq States$ is a finite set of states, $T \subseteq Threads$ is a finite set of threads, $A \subseteq Actions$ is a finite set of actions, $\Theta \subseteq A \times T$ denotes a total, function that maps each action to a thread, $\Delta \subseteq S \times \Theta \times S$ denotes a transition relation that maps a state and an action/thread pair to another state, and $q \in S$ indicates the initial state of P. For example, Figure 2.13 depicts how the LTS *Example* in Figure 2.11 would be defined.

We define a trace R of an LTS P to be a 3-tuple $\langle R_S, R_T, R_A \rangle$ where

- R_S is a sequence over S,
- R_T is a sequence over T,
- R_A is a sequence over A, such that $\mid R_T \mid = \mid R_A \mid = \mid R_S \mid -1$, $R_S(0) = q$, and

$$\forall i \in \{0 \ldots \mid R_T \mid -1\},\ (R_S(i),\ (R_T(i), R_A(i)),\ R_S(i+1)) \in \Delta.$$

For example, Figure 2.14 depicts a definition of the trace *ExampleTrace* from Figure 2.12.

Given a trace R of P, we define the following.

- The set of threads involved in R $threadSet(R) = \bigcup R_T(i)$, where i ranges from 0 to $\mid R_T \mid -1$. For example, $threadSet(ExampleTrace) = \{\ t_1,\ t_2\ \}$.

[7] Note that it is an action by t_2 that transitions t_1 out of the blocking state. This model is consistent with the way mutexes are implemented in POSIX Threads.

$$
\begin{aligned}
S &= \{\ s_0,\ s_1,\ s_2,\ s_3,\ s_4,\ s_5\ \} \\
T &= \{\ t_1,\ t_2\ \} \\
A &= \{\ t_1.\mathit{acquire}, \\
&\quad t_2.\mathit{acquire}, \\
&\quad t_1.\mathit{tryAcquireBlock}, \\
&\quad t_2.\mathit{tryAcquireBlock}, \\
&\quad t_1.\mathit{release}, \\
&\quad t_2.\mathit{release}, \\
&\quad t_1.\mathit{releaseUnblockT2}, \\
&\quad t_2.\mathit{releaseUnblockT1}\ \}
\end{aligned}
$$

$$
\begin{aligned}
\Theta &= \{\ (t_1.\mathit{acquire},\ t_1), \\
&\quad (t_2.\mathit{acquire},\ t_2), \\
&\quad (t_1.\mathit{tryAcquireBlock},\ t_1), \\
&\quad (t_2.\mathit{tryAcquireBlock},\ t_2), \\
&\quad (t_1.\mathit{release},\ t_1), \\
&\quad (t_2.\mathit{release},\ t_2), \\
&\quad (t_1.\mathit{releaseUnblockT2},\ t_1), \\
&\quad (t_2.\mathit{releaseUnblockT1},\ t_2)\ \} \\
\Delta &= \{\ (s_0,\ (t_1.\mathit{acquire}, t_1),\ s_1), \\
&\quad (s_0,\ (t_2.\mathit{acquire}, t_2),\ s_2), \\
&\quad (s_1,\ (t_1.\mathit{tryAcquireBlock}, t_1),\ s_3), \\
&\quad (s_1,\ (t_1.\mathit{release}, t_1),\ s_0), \\
&\quad (s_2,\ (t_2.\mathit{tryAcquireBlock}, t_2),\ s_4), \\
&\quad (s_2,\ (t_2.\mathit{release}, t_2),\ s_0), \\
&\quad (s_3,\ (t_1.\mathit{releaseUnblockT2}, t_1),\ s_2), \\
&\quad (s_4,\ (t_2.\mathit{releaseUnblockT1}, t_2),\ s_1)\ \}
\end{aligned}
$$

Figure 2.13: *Example* defined as a Multithreaded LTS.

$$
\begin{aligned}
R_S &= \langle\ s_0,\ s_2,\ s_0,\ s_1,\ s_3,\ s_2,\ s_0\ \rangle \\
R_T &= \langle\ t_2,\ t_2,\ t_1,\ t_2,\ t_1,\ t_2\ \rangle \\
R_A &= \langle\ t_2.\mathit{acquire},\ t_2.\mathit{release},\ t_1.\mathit{acquire},\ t_2.\mathit{tryAcquireBlock}, \\
&\qquad t_1.\mathit{releaseUnblockT2},\ t_2.\mathit{release}\ \rangle
\end{aligned}
$$

Figure 2.14: Definition of *ExampleTrace* based on a multithreaded LTS.

- Given a set of interesting actions $B \subseteq A$, the sequence of occurrences of those actions in R

  $$occurs_B(R) = \{(i,a) \mid a \in B \ \wedge \ (i,a) \in R_A\}.$$

  For example, given the set of block actions

  $$ExampleBlock = \{\ t_1.tryAcquireBlock,\ t_2.tryAcquireBlock\ \}$$

  and the set of unblock actions

  $$ExampleUnblock = \{\ t_1.releaseUnblockT2,\ t_2.releaseUnblockT1\ \},$$

  the sequence of block and unblock actions in $ExampleTrace$ are

  $$occurs_{ExampleBlock}(ExampleTrace) = \{\ (2,\ t_2.tryAcquireBlock)\ \}$$

  and

  $$occurs_{ExampleUnblock}(ExampleTrace) = \{\ (3,\ t_1.releaseUnblockT2)\ \},$$

  respectively.

- The sequence of threads in $threadSet(R)$ that reflects the order in which the threads are scheduled in R

  $$threadSchedule(R) = \{(i,t) \mid (i,t) \in R_T \ \wedge (i = 0 \ \vee \ R_T(i) \neq R_T(i-1))\}.$$

  For example,

  $$threadSchedule(ExampleTrace) = \{\ (0,t_2),\ (2,t_1),\ (3,t_2),\ (4,t_1),\ (5,t_2)\ \}.$$

Given a trace R of an LTS P, we define the following metrics.

- The number of threads involved in R is $|\ \mathit{threadSet}(R)\ |$. For example, the number of threads involved with $\mathit{ExampleTrace}$ is 2.

- Given the set of block actions $\mathit{Block} \subseteq A$ and unblock actions $\mathit{Unblock} \subseteq A$, such that $A_{\mathit{Block}} \cap A_{\mathit{Unblock}} = \varnothing$, the number of times threads block or unblock in R is

$$|\ \mathit{occurs}_{\mathit{Block}}(R) \cup \mathit{occurs}_{\mathit{Unblock}}(R)\ |.$$

  For example, threads in $\mathit{ExampleTrace}$ block/unblock a total of two times.

- The number of context switches that occur in R is $|\ \mathit{threadSchedule}(R)\ | - 1$. For example, four context switches occur in $\mathit{ExampleTrace}$.

Given a multithreaded LTS model, these metrics are trivial to compute for any trace R.

CHAPTER 3

EXPLORATORY STUDY: PLANNING AND EXECUTION

Little is known about how programmers cope with the challenges of debugging concurrent software in practice. We designed and conducted an exploratory study to discover the strategies and practices that successful programmers use. We took a qualitative approach, observing fifteen individual programmers as each performed a debugging task on a multithreaded server application, which we seeded with a defect. With this number of participants, we expected to observe a range of behaviors and outcomes. To reveal the goals and intentions of participants, we instructed them to "think aloud" as they engaged in the task. We collected rich qualitative data in the form of video recordings. Additionally, we used questionnaires to measure the general and task-specific knowledge possessed by participants both before and after the exercise. In this section, we describe the planning and execution of our exploratory study.

3.1 Study Planning

3.1.1 Participants

Fifteen students from a graduate-level formal methods course (CSE 814) at Michigan State University participated in the study. All participants happened to be male. Participation was voluntary, and participants were compensated with extra credit in the course.[1] To enroll in the course, students must have taken several programming-intensive courses. We expected participants to understand multithreaded programming at the level of an undergraduate operating-systems course. As preparation for the study, students received a 50-minute lecture on the multithreaded programming model used to implement the server application.

3.1.2 Study Materials

Materials in this study include a prestudy and a poststudy questionnaire, a workstation to perform the task with, the source code for the buggy multithreaded server, a stress-tester application to help reproduce the failure, and the debugging task instructions, which include a bug report describing the failure that arises from the defect.

Prestudy and Poststudy Questionnaires

The questionnaires measure knowledge using multiple-choice, true/false, and short-answer questions. We administered the prestudy questionnaire (or *prequestionnaire*) prior to performance of the task. The prequestionnaire measures participants' general knowledge of concurrent programming. We administered the poststudy questionnaire (or *postquestionnaire*) after performance of the task. The postquestionnaire measures participants' understanding of the program and of the seeded defect. Appendix A provides reproductions of these questionnaires.

[1] We offered alternative extra credit opportunities to students who did not want to participate.

Workstation

Participants performed the maintenance task on a computer workstation. We equipped the workstation with standard Web-browsing software (e.g., Internet Explorer and Firefox) and with software-development tools, which were familiar to the participant. For data collection, we outfitted the workstation with a microphone headset and Camtasia[2], a video-capturing application.

Buggy Multithreaded Application

We developed a multithreaded application, *eBizSim*, to use in the study. The application simulates an e-business server that processes requests from clients over the Internet. The eBizSim server accepts network connections from remote clients, receives requests from the clients over these connections, and simulates processing of the requests. The application is written in C++. We kept it small (229 SLOC) so that it is manageable for a user study in which the participants are seeing the code for the first time. To make the application more realistic, we based its architecture on the *reactor pattern* [107].

The server comprises multiple threads. Each thread either plays the role of *listener* or *handler*. A lone listener thread accepts client connections and places the requests received over these connections on a shared *request queue*. Meanwhile, multiple handler threads contend for requests by synchronizing on the request queue.

We seeded the eBizSim server with a design defect related to the proper use of *condition synchronization*[3] in the transfer of requests between the listener and handler threads. Condition synchronization is notoriously tricky and presents many potential pitfalls for programmers [105]. The defect manifests in a failure under certain timing and load con-

[2]`http://www.techsmith.com/camtasia/`

[3] Condition synchronization involves having a thread wait for some condition to be satisfied before proceeding to execute [76]. Waiting generally entails putting the thread "to sleep." A sleeping thread must be signaled by another thread to "wake up." When a waiting thread is awakened, it typically must recheck whether the condition is satisfied, and if not, it must wait again. Waiting and signaling are generally implemented using a mechanism called a *condition variable*.

straints. The stress-tester application provides a GUI control for adjusting the speed at which requests are sent to the server. Using the stress tester, it is often possible to reproduce the failure within a time frame of two to five minutes. However, we have never been able to reproduce the failure once the server runs without fail for more than five minutes. Thus, in practice, it is often necessary to restart the server multiple times to produce the failure. This design defect is representative of synchronization-related defects that are difficult to reproduce.

The defect stems from the way the listener and handler threads synchronize as they access the request queue. The request queue's elements are instances of a class `Request` (whose internal structure is not salient to this discussion). The request queue is managed by an object called the *pool*, which encapsulates and provides synchronized access to both the request queue and the thread pool. The pool defines a *mutex*[4] for each resource that it manages and a host of queue/pool-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. We concentrate on the operations that manipulate the request queue, as their code contained the defect.

The request queue is accessed through the operations `submit_request` and `retrieve_request`. Figure 3.1 depicts the implementations of these operations. Both methods acquire and release a mutex lock called `queue_lock_`. Thus, all calls to `submit_request` and `retrieve_request` execute under mutual exclusion. Moreover, calls to `retrieve_request` may block when the request queue is empty. The conditional blocking logic is implemented in lines 18–21 in `retrieve_request`, and the corresponding signaling logic (used to resume blocked threads when the blocking condition may have changed) is implemented in lines 6–9 in `submit_request`. The variable `nonempty_cond` refers to a *condition variable*, upon which threads may issue the operations `wait`, `signal`,

[4] A mutex is a mutual-exclusion lock that threads atomically acquire and release [76]. Only one thread at a time can hold a mutex. When a thread attempts to acquire an already-held mutex, the thread will block until the mutex is released by the holder.

```
void Pool::submit_request(Request* request)
{
  queue_lock_.acquire();
  request_queue_.push_back(request);

  if (queue_waiters_) {
    nonempty_queue_cond_.signal();
    --queue_waiters_;
  }

  queue_lock_.release();
}

Request* Pool::retrieve_request()
{
  queue_lock_.acquire();

  if (request_queue_.empty()) {
    ++queue_waiters_;
    nonempty_queue_cond_.wait();
  }

  if (request_queue_.empty()) {
    queue_lock_.release();
    return 0;
  }

  Request* request = request_queue_.front();
  request_queue_.pop_front();

  queue_lock_.release();
  return request;
}
```

Figure 3.1: Key synchronization methods in the eBizSim server.

and broadcast.[5] 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 submit_request method checks the value of this counter to decide whether it needs to signal the condition variable.

The defect occurs on line 18, where we replaced the line:

```
    while (request_queue_.empty()) {
```

with the line:

```
    if (request_queue_.empty()) {
```

Waiting in a while loop is a common pattern when programming with condition variables. We refer to this pattern as the *wait-in-while* idiom. To see how this defect may manifest in a failure requires reasoning about possible interactions between two handler threads and the listener thread during concurrent activations of retrieve_request and an activation of submit_request when the request queue is empty.

Instructions and Bug Report

The instructions given to participants ask them to play the role of an eBizSim maintainer in a scenario in which a user has reported a sporadic but troubling failure of the program. The participant's job is to fix the bug for the user. Figure 3.2 depicts the bug report filed by the user. In addition to a description of the failure, the bug report includes advice for reproducing the failure using the stress tester. Such detailed instructions for reproducing the failure are often difficult to come by but are equally often necessary to allow a vendor to reproduce a problem observed at a client site. For example, in the case of the buggy Therac-25 medical electron accelerator, a developer experimented for days to reproduce the failure [71].

[5] Our implementation is in C++ and uses primitives from the ACE toolkit. Readers familiar with Java can think of signal as analogous to notify and broadcast as analogous to notifyAll.

Bug Report / Change Request

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 3.2: Bug report provided to participants.

3.2 Execution

Prior to running the study, we recruited participants from the graduate formal-methods course. We sent an email to the enrolled students soliciting participation. We used extra credit in the course as an inducement. So that students would not be penalized for not participating, we offered an alternative means for earning the extra credit: an extra-credit homework assignment. Students indicated their participation by filling out a consent form. The form describes the activities involved with participation as well as the time commitment. The form reassured participants that any materials they produced for the study (e.g., filled-out questionnaires) will either be kept private or, if shared, be anonymized using randomly-assigned ID numbers.

3.2.1 Preparation

Several days prior to performing the maintenance task, participants received group instruction and completed the prequestionnaire. Group instruction took the form of a 50-minute lecture on concurrency constructs and their implementation using the ACE toolkit [108]. 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. Following the lecture, the participants completed the prequestionnaire on concurrency terminology and concepts.

3.2.2 Execution of Study Procedure

We scheduled participants for individual three-hour sessions, conducted in a private office. During the first 15–20 minutes, we introduced participants to the equipment and environment, and calibrated the audio-recording application. Next, participants engaged in think-aloud on a warm-up task. The task involved correcting a defect in the implementation of a bubblesort procedure [33]. *Prompters* trained in the think-aloud method accompanied the participants as they engaged in the maintenance task. When a participant would fall silent, the prompter would ask him to "please, keep talking." Following the warm-up, we gave participants a brief tour of the directories containing the eBizSim software and provided them with the bug report. We also provided participants with scratch paper, a brief guide to concurrency constructs in ACE, and a C++ manual [118]. Participants were permitted to browse the Internet as they deemed necessary while performing the task.

Participants were allotted up to 150 minutes to complete the task. Those who completed the task sooner could stop the session at that time. Immediately following the sessions, participants took the postquestionnaire, which was designed to evaluate their comprehension of the eBizSim server and the defect.

CHAPTER 4

EXPLORATORY STUDY: STRATEGIES AND CODING SCHEMES

The goal of this study was to find relationships between the strategies and practices of programmers (*strategies* for short), and the level of success on task. Having collected the think-aloud data, we identified a set of strategies, which we deemed to be of interest. We drew this set from two sources. First, we performed detailed reviews (and re-reviews) of the think-aloud data, identifying distinct behaviors that the participants exhibited during their sessions. Second, we surveyed the literature for strategies that were claimed to lead to success on programming tasks. In the end, we singled out six strategies for study. In this section, we describe each of these strategies in turn and explain how we code them in our data.

4.1 Failure-Trace Modeling

We discovered participants using a previously-undocumented strategy, which we call *failure-trace modeling*, for determining whether a hypothetical error state is reachable by some execution of the program. The strategy aims to produce a *failure trace*, which demonstrates how the system transits among various internal states, at least one of which is a clear error state, up to the point of failure. Following the strategy, the programmer first formulates an *error suffix*, which models a fragment of a candidate failure trace and ends in a clear error state. Next, the programmer attempts to verify that the error suffix is *feasible*—that is, consistent with an actual execution trace. The general process and the models produced are best illustrated by example. We use a multithreaded sequence diagram to depict these models.[1]

4.1.1 Example

Figure 4.1 depicts an (infeasible) candidate error suffix, as articulated by one of the participants in our study. Here, two handler threads (denoted h_1 and h_2) concurrently attempt to retrieve a request from an empty request queue. Because the queue is empty, both threads wait on the condition variable `nonempty_cond` (abbreviated `qNonempty` in the figure). Shortly thereafter, the listener thread (denoted l) invokes a `submit_request` operation, which adds a request (denoted `r`) to the queue and, according to the figure, invokes a `broadcast` operation on `qNonempty`. In response to the broadcast, both handler threads resume and attempt to reacquire the lock. Here, h_1 acquires the lock and is able to proceed, after which it pulls `r` off of the queue and releases the lock. Thread h_2 acquires the lock next and proceeds. However, because the queue is once again empty, the `retrieve_request` method returns 0, which causes h_2 to enter an error state.

The notational conventions in Figure 4.1 deserve additional explanation. To distinguish type/class names (e.g., `Pool`) from role names (e.g., *Handler*), we render the latter in italics. To indicate that the queue is empty at the start of the suffix and show how it mutates during the trace, the lifeline of the pool object (denoted `rhp`) is adorned with object states (e.g., "$q = ()$"), which are depicted inside roundtangles that are centered atop the corresponding activation bars. Additionally, to reduce nonessential clutter, messages corresponding to method invocations are rendered abstractly, without showing all the objects involved, but showing the thread that invokes them. For example, calls to `retrieve_request` are ac-

[1] See Section 2.4.2 for a description of the notation.

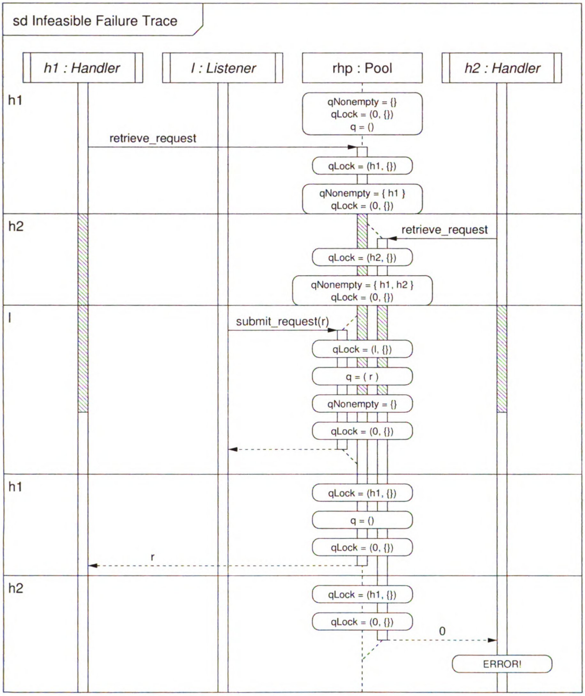


Figure 4.1: Strongly-articulated, yet infeasible model of an error trace.

tually made from within activations of another pool method, `dispatch_request` (omitted for clarity), and a more detailed model would show the `retrieve_request` messages emanating from activations (of `dispatch_request`) that are attached to the pool's lifeline. As the figure does not depict activations of `dispatch_request`, we instead show the messages emanating from the lifelines of the thread executing the elided activations.

A programmer might construct such a model to use in determining if the failure is a result of a call to `retrieve_request` that returns null. The model documents the initial state of the relevant object (i.e., the pool), and it describes the number and role(s) of the interacting threads. Additionally, the activation predicates are consistent with the semantics of available synchronization primitives (in this case, with `acquire`, `release`, `wait` and `broadcast`). Such a model represents a candidate error suffix. If the initial state of the model is consistent with a reachable program state, and if the actions and state transitions in the model are consistent with the code, then the model describes a feasible trace that ends in an error and that therefore exhibits a defect.

4.1.2 Feasibility Analysis

A candidate error suffix exhibits an actual error only if it is feasible. Such a model seldom begins in the initial state of the program, and it usually elides many details. Thus, when reasoning about thread interactions, the programmer can easily construct a model that is not feasible. This holds especially true in the context of maintenance where the programmer may lack a global view of the program. In fact, the model in Figure 4.1 is not feasible because the code executed by the listener thread uses `signal` rather than `broadcast` to notify a handler thread of a request in the queue. Thus, only one of the handlers will be awakened during the activation of `submit_request`, whereas the model depicts both as being awakened.

Generally speaking, a deep analysis is required to decide whether a candidate error suffix is feasible. Given an error suffix, this analysis can be performed by (1) instantiating the abstract components with concrete objects, activations, threads, and states, and (2) checking that the initial state of the instantiated model is consistent with a state reachable from the initial state of the program. Instantiating the abstract components involves binding the abstract actors (depicted as active objects in the figure) to activations of operations on concrete objects by concrete threads, and verifying that the sequence of activation- and object-state transitions depicted in the error suffix is consistent with the implementation of the associated operations in the code. Feasibility analysis may involve global reasoning and is generally difficult.

4.1.3 Strength of Articulation

In studying the failure-trace modeling strategy, we discovered distinct differences in the quality (i.e., clarity and distinctness) of the models articulated by our participants. Some referred to threads only in the abstract, whereas others also identified which thread acted as the listener and which threads acted as handlers. Additionally, some participants articulated the interactions that might culminate in an error with sufficient detail that we could construct an error suffix, such as is depicted in Figure 4.1, whereas others did not articulate necessary information for formulating an error suffix. For example, a participant might say that an operation is invoked, but not say which thread invokes the operation. Or, in contrast, he might not say when a thread acquires (releases) a lock or when it waits (awakens from a wait).

We classify a model as *strongly articulated* if it satisfies three properties. First, it must describe a collaboration among distinct actors and objects with well-defined synchronization states. That is, it should always be clear which actors, if any, are blocking, are holding a mutex lock, are waiting on a mutex lock, and are waiting on a condition variable. Second, the model must describe how actions by the actors cause the objects and other actors to transition among these synchronization states. For instance, an actor must invoke `acquire` on a mutex lock to become holder of the lock. In a strongly articulated model, an actor should never become holder of a lock without having made the necessary call to `acquire`. Third, it must be articulated (e.g., drawn or verbalized) in sufficient detail to enable formulation of a well-formed UML sequence diagram, such as the one in Figure 4.1.

4.1.4 Coding Scheme

We looked for evidence of the failure-trace modeling strategy using four attributes: mod, err, art, and succ. These attributes indicate various levels of modeling. The mod attribute indicates that the participant gave evidence of modeling some interaction among multiple threads synchronizing over shared data. The err attribute indicates that he modeled an interaction that manifests in an error state (although the model may represent an infeasible error suffix). The art attribute indicates that he produced a model that was strongly articulated. The succ attribute indicates that he produced a strongly-articulated model that accurately describes the error that led to failure. We also considered coding for feasibility checking; however, no participant exhibited behavior that we could associate with this activity.

We count as evidence of modeling any description (be it a verbalization or a drawing) that supposes the existence of two or more threads and one or more shared objects and that proceeds to develop a sequence of actions by and among these entities. Clearly, err, art, and succ imply mod. However, a modeled interaction need not involve an error state (or indeed have anything to do with the failure in question), and modeling may or may not be strongly articulated. Moreover, err does not imply art, and art does not imply err. Additionally, succ implies but is not equivalent to err $\wedge$ art.

4.2 Breadth-First Approach to Diagnosis

Vessey [123] previously found 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. When debugging concurrent software, we expect that programmers will tend to have difficulty reducing the space of plausible hypotheses regarding the cause of the defect because the usual techniques for doing so, cyclic debugging and dependence analysis, are ineffective. Thus, programmers are confronted with a large space of competing hypotheses to analyze. However, analysis of a hypothesis is often laborious and time consuming because it involves reasoning about complex thread interleavings. Therefore, programmers should invest effort in such analyses sparingly and deliberately. The prior work refers to such behavior as *breadth-first problem solving* and suggests that experts apply it during debugging [123].

Breadth-first refers to the order in which hypotheses are analyzed. Initially, the programmer forms a small set of hypotheses (possibly only one). The analysis of a hypothesis (the parent) typically yields new, more specific and detailed hypotheses (children). Thus, the parent-child relationships between hypotheses take on a tree-like structure where the leaves are hypotheses that do not warrant further refinement. The breadth-first approach is roughly analogous to a breadth-first search in graph theory. That is, hypotheses at higher levels of the tree(s) are generally analyzed before those that are lower. Note that the approach does not follow a strict breadth-first search in the algorithmic sense. As a consequence of this approach, competing hypotheses are considered before the effort is invested to analyze and refine any one in particular. In contrast, a depth-first approach tends to pursue a linear path through the tree, ignoring competing hypotheses until no alternative remains.

4.2.1 Modeling Hypothesis Elaboration and Refinement

We investigated the application of breadth-first problem solving by studying 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 model of the space of hypotheses that could pertain to the failure in our study and of the parent-child relationship among these hypotheses. To construct such a model, we used an extended form of software fault trees, which model the causal relationships between the possible events that could lead to a failure. As a programmer debugs a defect, he formulates hypotheses that correspond to events in the fault tree. Thus, as he generates hypotheses, he can be seen as elaborating the fault tree. By associating utterances and actions in the think-aloud data to events and patterns of elaboration in the fault tree, we checked which participants applied the breadth-first approach, whether there was a relationship between the approach and success, and whether concurrency hindered the approach. The remainder of this section provides a brief introduction to the model we developed, including a proposed extension to the standard fault-tree notation. Although a formal treatment of this proposed extension is beyond the scope of this work, we suggest how it might be formalized in terms of standard compiler abstractions (page 70). The uninterested reader may safely skip over this speculative material.

Extended Fault-Tree Notation

We explain our fault-tree notation by example. Figure 4.2 depicts a portion of the fault tree for the failure under study. We provide the complete fault tree in Appendix C. Each box (or *node*) in the tree represents an *event*, which is an observation about the properties of system objects made at some moment in time. Although events here are informal, we have an idea about how we might formalize them (page 70). However, fully developing this formalization is beyond the scope of this dissertation. Each event has a label (e.g., *root* and M_1) so that it may be referred to by name. The label is listed in the upper compartment of the event node.

A node in the fault tree comprises a *control point*, a *state predicate*, and an optional *timing constraint*. These parts are listed on separate lines in the lower compartment of the node. The control point represents the location of a particular thread's execution with respect to the source code. Appendix A provides a listing of the source code for the eBizSim application. Each event is observed at a distinct control point. For example, node M_1's

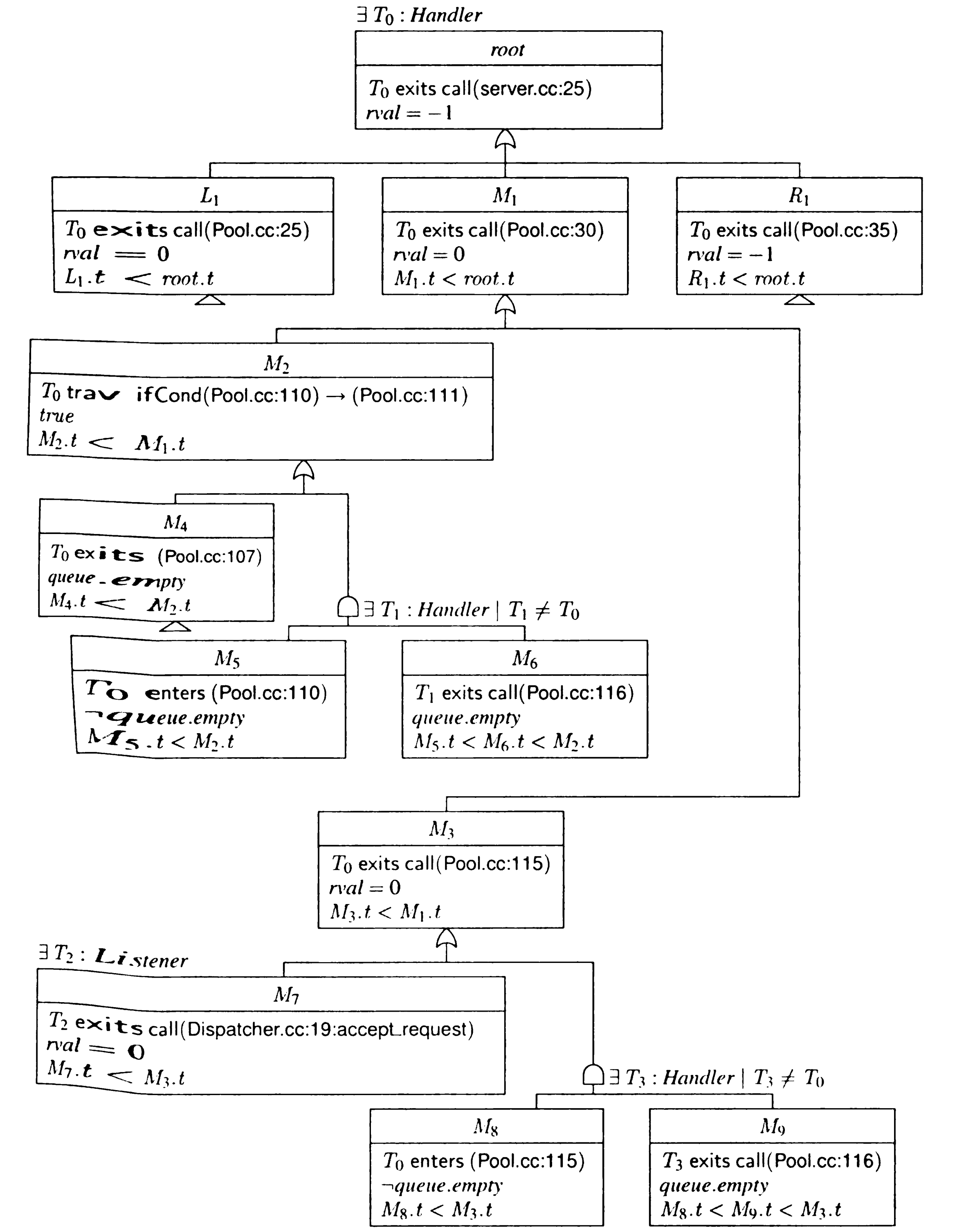

$\exists T_0 : Handler$
root
T_0 exits call(server.cc:25)
$rval = -1$
L_1
T_0 exits call(Pool.cc:25)
$rval = 0$
$L_1.t < root.t$
M_1
T_0 exits call(Pool.cc:30)
$rval = 0$
$M_1.t < root.t$
R_1
T_0 exits call(Pool.cc:35)
$rval = -1$
$R_1.t < root.t$
M_2
T_0 trav ifCond(Pool.cc:110) → (Pool.cc:111)
true
$M_2.t < M_1.t$
M_4
T_0 exits (Pool.cc:107)
queue_empty
$M_4.t < M_2.t$
$\exists T_1 : Handler \mid T_1 \neq T_0$
M_5
T_0 enters (Pool.cc:110)
$\neg queue.empty$
$M_5.t < M_2.t$
M_6
T_1 exits call(Pool.cc:116)
queue.empty
$M_5.t < M_6.t < M_2.t$
M_3
T_0 exits call(Pool.cc:115)
$rval = 0$
$M_3.t < M_1.t$
$\exists T_2 : Listener$
M_7
T_2 exits call(Dispatcher.cc:19:accept_request)
$rval = 0$
$M_7.t < M_3.t$
$\exists T_3 : Handler \mid T_3 \neq T_0$
M_8
T_0 enters (Pool.cc:115)
$\neg queue.empty$
$M_8.t < M_3.t$
M_9
T_3 exits call(Pool.cc:116)
queue.empty
$M_8.t < M_9.t < M_3.t$

Figure 4.2: Part of the fault tree for the eBizSim failure.

control point, "T_0 exits call(Pool.cc : 30)", specifies that the M_1 event is observed when thread T_0 exits the call (to `retrieve_request_from_queue`) on line 30 of the file Pool.cc. A thread's existence must be declared before the thread can be referred to in a control point. We describe thread declarations below.

The state predicate expresses a property of the system's state that is true when the event is observed. For example, event M_4 has the state predicate "$queue.empty$", which indicates that the request queue is empty when the event is observed. The state predicate "*true*", as seen in M_2, indicates no particular property of the system state. When the event's control point is the exit from a function, we use the keyword *rval* in the state predicate to refer to the value returned by the function call. For example, the M_1 event's state predicate indicates that the value returned by the call on line 30 of Pool.cc is 0. Note that *rval* is meaningful only if the event's control point is the exit of a call.

The timing constraint specifies the order in which the event occurs relative to other events. For expressing timing constraints, we use $M.t$ to denote the time event M occurred. For example, the timing constraint in the event node M_1 indicates that event M_1 occurs before event $root$. As another example, the timing constraint in event node M_9 indicates that event M_9 occurs after event M_8 and before event M_3.

Event nodes may be connected by edges, which indicate parent-child relationships between the nodes. The event nodes and edges form a hierarchical tree structure. In the figure, parent nodes always appear vertically higher than their children. If a child event occurs, its parent event must occur. That is, the occurrence of a child *enables* its parent. The fault tree has a distinct root event, which represents the failure event. The leaf nodes of the fault tree represent events whose likelihood of occurring can be assessed without further refinement. The analyst decides which events are appropriate to represent as leaf nodes.

Edges in the fault tree may pass through logic gates. An OR gate indicates alternative enabling events. For example, the edges that connect the node *root* with its children L_1, M_1, and R_1 pass through an OR gate. Thus, the occurrence of any one of the child events enables the *root* event. An AND gate indicates a conjunction of multiple child events, all of which must occur together to enable the parent. To make the notion of *together* more precise, we conjoin the time constraints of the child events. For example, the edges that connect M_5 and M_6 to their parent M_2 pass through an AND gate. Thus, an occurrence M_5 followed by an occurrence of M_6 enables M_2.

A node or logic gate may be annotated with a thread declaration. The declaration has the form "$\exists$ *ThreadName* : *ThreadRole* | *Constraint*", where the *ThreadRole* and *Constraint* parts are optional. It expresses that there exists a thread named *ThreadName* that plays the role of *ThreadRole* and adheres to some *Constraint*. The *Constraint* is typically used to distinguish the declared thread from other previously declared threads. The visibility of a *ThreadName* in the fault tree is limited to the subtree rooted by the node or gate that bears the thread declaration. For example, node *root* is annotated with a declaration for the thread T_0 which plays the role of a *Handler* in the system. Note that the root event must always declare a thread because the event's control point must have an associated thread. As another example, the AND gate connected to nodes M_5 and M_6 is annotated with a declaration for the thread T_1 that plays the role of *Handler* and is a different thread than T_0.

Fault-Tree Example

Having explained our fault-tree notation, we now provide a high-level description of the fault tree in Figure 4.2. The root event in the tree represents an error state that manifests in the failure described in Figure 3.2. The error occurs at time *root.t* when a thread T_0 observes an invocation of the `dispatch_request` method (line 25 of server.cc) returning -1. The code for `dispatch_request` (lines 21–41 of Pool.cc) can return -1 in any one of three statements. Nodes L_1, M_1, and R_1 represent events that will cause one of these statements to be executed. Evaluation of retrieve_handler at line 25 of Pool.cc could return 0 (L_1), thereby causing `dispatch_request` to return -1 (line 27). Nodes M_1 and R_1 can be understood similarly. Figure 4.2 shows the M_1 subtree, which is the alternative that actually caused the seeded defect. Triangles indicate where subtrees have been elided.

The M_1 subtree explains how an invocation of `retrieve_request` could have returned 0. Either evaluation of the if condition on line 110 of Pool.cc succeeded (event M_2), or the method invocation on line 115 returned 0 (M_3). Likewise, the M_2 subtree explains how the if condition on line 110 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 110 believing the queue is non-empty, but another thread empties the queue before T_0 can execute the if statement on line 110. The remaining events in Figure 4.2 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.

†Proposed Formalization of Fault Trees

To represent events more precisely, we propose to formalize them in terms of control flow graphs (CFGs), which we previously introduced in Section 2.1.3. Here, we assume CFG nodes are labeled with instructions in *three-address code*, which comprises statements similar to those in an assembly language [1]. The three-address code language includes assignment statements, simple arithmetic statements, and control-flow statements (e.g., `goto`). A three-address statement typically contains no more than two operands and one result (thus, three addresses). Compound statements in the source, such as "`x = a * b + c / d`", engender a sequence of three-address statements, such as:

```
temp1 = a * b
temp2 = c / d
x = temp1 + temp2
```

Therefore, a single statement in the program may engender many nodes and edges in the CFG.

Given a CFG of the program, control points now represent the location of a particular thread's execution with respect to the CFG. We distinguish three types of control points at which an event may be observed. For a thread T and CFG nodes N and N', an event may occur (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. In contrast, T exits N is coincident with termination of the instruction at N.

To describe the CFG node of a control point, we use an informal *CFG-node designator*. At the minimum, the designator lists the source file and line number associated with the CFG node. For example, "Pool.cc:110" indicates the CFG node associated with line 110 of the source file Pool.cc. In cases where the line of code engenders multiple CFG nodes, we have two ways to disambiguate the CFG node. First, we prefix the CFG-node designator with a *qualifier* to indicate the type of instruction in the CFG node being designated. For instance, the control point for the event node *root* is given as "T_0 exits call(server.cc:25)," which indicates that the event is observed when the thread T_0 exits the CFG node that contains the `call` instruction associated with the function call on line 25 of server.cc. In this case, there is only one function call on that line, `dispatch_request`, so no further disambiguation is necessary. In addition to the call qualifier, we have an ifCond qualifier that designates the CFG node containing the `if-goto` instruction that branches to the CFG node designated by the target of a trav specifier. The second way to disambiguate CFG nodes is by appending the concrete syntax of a program statement or expression after the file and line number in the CFG-node designator. For example, the control point for event node M_7 is given as "T_2 exits call(Dispatcher.cc:19:accept_request)," which indicates that the event is observed when the thread T_2 exits the call to the function `accept_request` on line 19 of the file Dispatcher.cc. In this case, there are two function calls on line 19, so we disambiguate which call we mean by adding the name of the function (i.e., `accept_request`) to the CFG-node designator.

4.2.2 Coding Scheme

We looked for evidence of a breadth-first approach with one attribute: bread. The attribute indicates that a participant attempted to use a breadth-first approach to diagnosing the defect. To perform the encoding, we associated each plausible hypothesis that a participant discovered or analyzed with an event in the fault tree. We say that a participant *discovered* an event if he somehow verbalized the control point, state predicate, and/or time constraint associated with the event. We say he *analyzed* the event if he attempted to determine how it could be enabled. In some cases, event discovery and analysis are clearly articulated in the protocols. For instance, participant 06 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 participant 09 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 participant 09 discovered M_4 as an enabler while analyzing M_2. We coded a participant as bread if he analyzed competing events before investing heavily in a particular subtree, and if he analyzed a sufficiently large number of the events in the fault tree down to depth two.

4.3 Systematic Comprehension

Littman and colleagues [73] claim that the use of a systematic comprehension strategy, which involves reading the entire source in a thorough and structured way, predicts success on maintenance tasks, such as debugging. We previously described this work in Section 2.3. In contrast to a systematic strategy, an as-needed strategy involves reading only those parts of the code that are believed to relevant to the task at hand and ignoring the rest and does not predict success. They claim that the systematic strategy leads to the development of a strong mental model of a program, which they define as containing static and causal knowledge about the program. Static knowledge refers to an understanding of a program's functional components (e.g., roles, classes and methods), whereas causal knowledge refers to an understanding of how the functional components interact at run time. This prior work also claims that both strategies produce sufficient static knowledge, but the as-needed comprehension strategy produces weaker causal knowledge than the systematic strategy.

By definition, the systematic strategy involves the programmer performing extensive mental execution of the data flow paths between subroutines [73]. The programmer proceeds by starting at the main routine and following the control and calling structure of the subroutines. He must "actually imagine the behavior of the program as if it were running in time," thereby providing him with "causal knowledge about the order of actions in the program" [73]. A programmer needs causal knowledge to diagnose and fix the defect in the eBizSim program, which manifests as an erroneous 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 [116]. Synchronization logic typically manifests as a delocalized plan. Although others [59, 124, 100] have expressed doubt about the effectiveness of systematic comprehension for large software systems, the eBizSim program is small enough to be read in its entirety within the time frame of the participant sessions.

A strong mental model contains both static and causal knowledge about the program. 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 multithreaded 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.

Causal knowledge pertains 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.

4.3.1 Coding Scheme

We looked for evidence of a systematic comprehension strategy with one attribute: syst. 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 defect. 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 starting 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.

4.4 Cyclic Debugging

Cyclic debugging is a widely-used to technique for diagnosing a defect. The technique involves repeatedly executing a failing run of the buggy program to observe the program's internal state as it executes. The internal state is made observable with diagnostic print statements or with the help of a debugger. With each iteration, the programmer adjusts which state information is observable (e.g., by adding/modifying diagnostic print statements) until the output sufficiently explains how the failure occurs. Although the technique may be effective in the context of sequential software, it is ineffective when failures are difficult to reproduce, which is the case with the program in our study. Section 2.1.1 describes cyclic debugging in detail.

4.4.1 Coding Scheme

We looked for evidence of cyclic debugging using three attributes: inst, refine, and found. The attributes record various levels of the technique. The inst attribute indicates that a participant instrumented the code by adding diagnostic print statements. The refine attribute indicates that a participant refined diagnostic statements to expose more detailed internal-state information based on the diagnostics provided by a previous failing run of the program. The found attribute indicates that a participant produced a failing trace whose diagnostics indicate the error state that led to failure and the sequence of synchronization events that caused the program to enter this state. Clearly, each of refine and found implies inst. However, found does not imply refine, even though we expect found to be highly correlated with refine. We collected refine to measure the level of investment in this technique—that is, whether the participant taking an iterative approach to narrowing down the source of the defect. The criteria for recording found are quite strong: a run has to produce a failure trace whose diagnostics show at least two handler threads waiting, the listener thread arriving and signaling both of them, one handler waking and emptying the queue, and the second waking to an empty queue.

4.5 Pattern Matching

Ben-David Kolikant [11] observed that novices tend to us a pattern-based technique to solve synchronization problems. Novices use memorized patterns of code to reason about how synchronization code should be written. Such patterns include programming idioms, such as the wait-in-while idiom (described in Section 3.1.2). The pattern-based technique is effective for programs that build using familiar patterns. Unfortunately, novices often do not understand the synchronization mechanisms that underlie the patterns and have difficulty in situations where familiar patterns are not applicable.

4.5.1 Coding Scheme

We looked for evidence of pattern matching using one attribute: pat. The attribute indicates that a participant recognized the violation of the wait-in-while pattern. We accepted utterances that display familiarity with the idiom, such as "seems like a problem we've seen in class before, where if you don't embed your condition variable in a [while]," as evidence of this attribute.

4.6 Tweaking the Code

We also observed some participants appealing to what seemed to be little more than luck. They tweaked the synchronization code without explanation or expectation of success, seemingly in an attempt to luck into a fix. After making a tweak, participants would often re-execute the program to see if the change fixed the defect. However, this testing has an inherent weakness because the absence of a failure does not constitute proof of success in the context of concurrent software. Moreover, the difficulty of reproducing the failure should make the futility of this strategy evident.

4.6.1 Coding Scheme

We looked for evidence of this behavior with one attribute: luck. We recorded luck when a participant modifies the synchronization logic or other logic that could affect thread schedules and then executes the program to "see what happens." Examples include reordering the lines:

```
nonempty_queue_cond_.signal();
--queue_waiters_;
```

in `submit_request`, and removing all of the code involving the `queue_waiters_` variable. In each of these cases, the participant uttered something to indicate a lack of any real expectation that the tweak would solve the problem—for example, "if that actually was the problem, I'll be upset."

CHAPTER 5

EXPLORATORY STUDY: ANALYSIS AND DISCUSSION

Having identified strategies of interest and developed coding schemes for them, we analyzed the think-aloud data. The main goal of our analysis was to identify the strategies and practices that successful participants tended to use. Section 5.1 details how we performed the analysis and the results that emerged. Section 5.2 discusses our interpretation of the results and the potential limitations of our study.

5.1 Analysis

Prior to the analysis, we identified five distinct attributes related to success, three related to diagnosis and two related to correction, and developed coding schemes for them. The three attributes related to success at diagnosing the defect are loc, rat, and conf. We record loc for each participant who was able to narrow his search to the code segment that contains the seeded defect. This attribute is recorded without regard to whether the participant could explain why the code was buggy or whether the participant was ultimately able to correct the defect. We record rat for each participant who was able to explain why the buggy code is defective. As evidence of rat, we look for verbalizations, such as: "when `wait` returns, the queue is not guaranteed to be non-empty," or, "I now see how the queue can be empty when the thread exits the if block." Clearly, rat implies loc, but the converse is not true. The attribute conf indicates an expression of high confidence that the participant has correctly diagnosed the defect. As evidence of conf, we look for explicit declarations of confidence such as "I am really confident that I identified the problem correctly," and exclamations that implicitly indicate confidence such as, in reference to a defect diagnosis, "that's gotta be it!" Notice that this attribute indicates a participant's opinion of his performance and need not reflect real success. The two attributes related to success at correcting the defect are fix and noNew. We record fix for each participant who successfully fixed the seeded defect, which essentially means that he changed the `if` block into a `while` block in the `retrieve_request` method. We record noNew for each participant who fixed the seeded defect as indicated and did not introduce a new defect.

Our analysis comprised two main steps. First, using the coding schemes we developed, we encoded the think-aloud data for the strategy and success attributes. At least two researchers individually encoded each session. Then, they compared their encodings, resolved any differences, and, if necessary, refined the coding scheme(s). Table 5.1 summarizes the attributes and their definitions, and Table 5.2 lists the results of our encoding ("+" indicates true and "-" indicates false). Second, we performed cross-case analyses to assess the relationship between each strategy and success. We distinguished five levels of success in our analyses: loc, rat, rat $\wedge$ conf, fix, and noNew. For each level of success, we partitioned the participants into two groups: one that achieved the level and one that did not. We looked for significant differences between these groups with respect to each strategy attribute. We used the Fisher exact probability test [110] to assess the likelihood that an observed difference occurred by random chance. The Fisher test is appropriate for small sets of dichotomous data such as ours. We set the alpha for the test at 0.05. Table 5.3 summarizes the results of our Fisher tests, depicting the p-values that are at or below the alpha in bold.

In the remainder of this section, we describe in detail the results of our analyses. First,

Table 5.1: Attributes of success and participant behavior.

Attribute	Description
loc	Participant identified that the failure is triggered by returning zero on line 22 in `retrieve_request` (see Figure 3.1).
rat	Participant provided correct rationale to explain why the buggy code is defective.
conf	Participant expressed high confidence that he had correctly diagnosed the defect.
fix	Participant replaced the buggy `if` statement with a `while` statement in `retrieve_request`.
noNew	Participant fixed the defect without introducing any new defects.
mod	Participant gave evidence of modeling some interaction among multiple threads synchronizing over shared data.
err	Participant gave evidence of modeling an interaction that manifests in an error state (although the model may represent an infeasible error suffix).
art	Participant produced a model that was strongly articulated.
succ	Participant produced a strongly-articulated model that accurately describes the error that led to failure.
bread	Participant took a breadth-first approach to diagnosing the defect (as opposed to a depth-first approach).
syst	Participant used a systematic comprehension strategy (as opposed to an as-need strategy).
inst	Participant instrumented the code by adding diagnostic print statements.
refine	Participant refined diagnostic statements to expose more detailed internal-state information based on the diagnostics provided by a previous failing run of the program.
found	Participant produced a failing trace whose diagnostics indicate the error state that led to failure and the sequence of synchronization events that caused the program to enter this state.
pat	Participant recognized violation of a pattern commonly used to implement condition synchronization in monitors.
luck	Participant tweaked the synchronization code without explanation or expectation of success, seemingly in an attempt to luck into a fix.

Table 5.2: Assignment of attributes to participants.

Attribute	Participant															Frequency
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
loc	+	-	+	+	-	+	+	+	+	+	+	+	+	-	+	12
rat	+	-	-	-	-	+	+	+	+	+	-	+	-	-	+	8
conf	-	-	-	+	-	+	+	-	+	+	-	-	-	-	+	6
fix	+	-	-	-	-	+	+	+	+	+	-	+	-	-	+	8
noNew	+	-	-	-	-	+	-	+	+	+	-	+	-	-	-	6
mod	+	+	-	+	-	+	+	+	+	+	+	+	-	-	+	11
err	+	+	-	+	-	+	+	+	+	+	-	-	-	-	+	9
art	-	+	-	-	-	+	+	+	+	+	+	+	-	-	+	9
succ	-	-	-	-	-	+	+	-	+	-	-	-	-	-	+	4
bread	+	-	-	-	-	+	-	+	+	+	+	+	+	+	+	10
syst	-	-	+	-	+	+	+	+	+	+	+	+	-	-	+	10
inst	+	+	+	+	-	+	+	+	+	+	+	+	+	+	-	13
refine	+	+	+	+	-	+	+	+	+	-	+	+	-	+	-	11
found	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
pat	-	-	-	-	-	+	+	-	+	+	-	+	-	-	+	6
luck	+	+	+	+	+	-	-	-	+	-	+	+	+	+	-	10

Table 5.3: Statistical results of cross-case analyses (p-values).

Attribute	loc	rat	rat $\wedge$ conf	fix	noNew
mod	0.1538	**0.0256**	0.2308	**0.0256**	0.1032
err	0.5253	**0.0406**	0.0889	**0.0406**	0.2867
art	0.5253	**0.0406**	0.0889	**0.0406**	0.2867
err $\wedge$ art	1.0000	**0.0406**	**0.0070**	**0.0406**	0.3147
succ	0.5165	0.0769	**0.0037**	0.0769	1.0000
bread	0.5055	0.1189	0.6004	0.1189	0.0889
syst	0.5053	0.1189	0.1009	0.1189	0.5804
inst	0.3714	1.0000	1.0000	1.0000	0.4857
refine	1.0000	1.0000	0.5604	1.0000	0.6043
pat	0.2286	**0.0070**	**0.0020**	**0.0070**	0.1357
luck	0.5055	**0.0256**	**0.0170**	**0.0256**	0.5804

we provide descriptive statistics regarding the levels of success. Second, we describe the results of the cross-case analyses for each strategy. Some strategies warranted additional analyses, which we describe as well.

5.1.1 Levels of Success

Figure 5.1 depicts the frequencies of the various levels of success in the data. Twelve of the fifteen participants succeeded in localizing the defect to the correct segment of code (loc). Given this strong majority, we did not find a statistically significant relationship between any attribute and loc (see Table 5.3). Because of the lack of interesting results, we will not discuss analyses involving loc further in this chapter. Only eight participants were able to explain the defect (rat). As shown in Table 5.2, these same eight participants were also able to fix the defect (fix). Five of the eight participants who explained the defect did so with confidence (rat $\wedge$ conf). Six of the eight participants who fixed the defect did so without introducing any new defects (noNew).

5.1.2 Failure-Trace Modeling

We performed cross-case analyses of our data to look for relationships between the use of failure-trace modeling and various measures of success. Figure 5.2 depicts the frequencies of each modeling attribute among the participants at each level of success. The first five rows of the Table 5.3 provide the statistical results of the comparisons.

We compared the participants who correctly diagnosed the defect with those who did not. We found that participants in the first group were significantly more likely to model (mod) ($p < 0.05$), to produce an error suffix (err) ($p < 0.05$), and to produce a strongly-articulated model (art) ($p < 0.05$). Furthermore, all the participants who successfully constructed a failure trace (succ) correctly diagnosed the defect. These results suggest that there is a strong relationship between modeling and correctly diagnosing the defect.

We compared the participants who correctly diagnosed the defect with confidence (rat $\wedge$

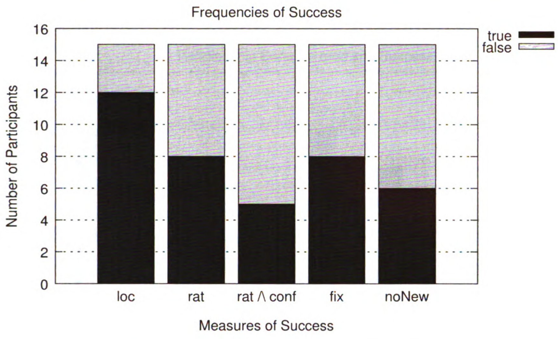
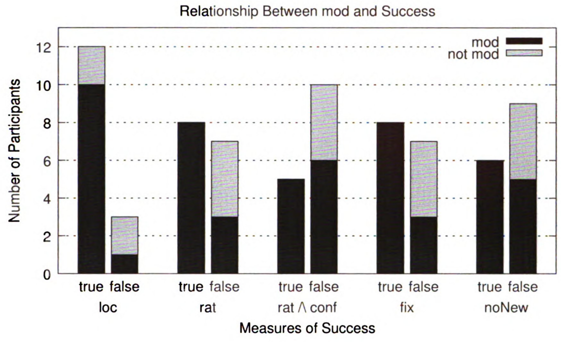


Figure 5.1: Frequencies of various measures of success.

conf) with those who did not. We found that the first group was significantly more likely to produce both an error suffix and a strongly-articulated model (err ∧ art) ($p < 0.01$), and to successfully model a failure trace (succ) ($p < 0.01$). This result suggests that participants who applied the failure-trace modeling strategy more thoroughly were more likely produce a (correct) diagnosis that they were confident in.

We analyzed the data for relationships between modeling and fixing the defect. We compared the participants who fixed the defect (fix) with those who did not. We found that the first group was significantly more likely to engage in failure-trace modeling (mod, err, art, and err ∧ art) ($p < 0.05$). As with diagnosis, all participants who successfully produced a failure trace (succ) fixed the defect. However, modelers seem to have been susceptible to introducing new defects. We found this by comparing the participants who produced a correct solution without introducing new defects (noNew) to those who did not. Although a greater proportion of the first group engaged in modeling at all levels, the difference between the groups did not rise to statistical significance for this small sample. One possible



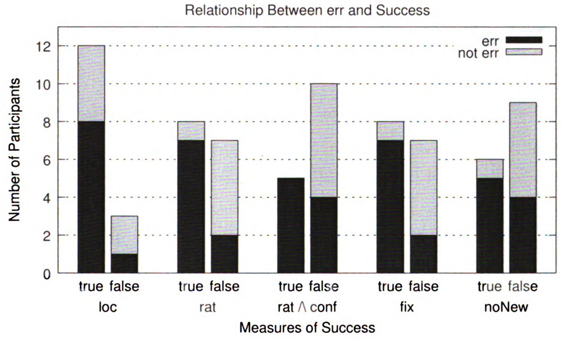
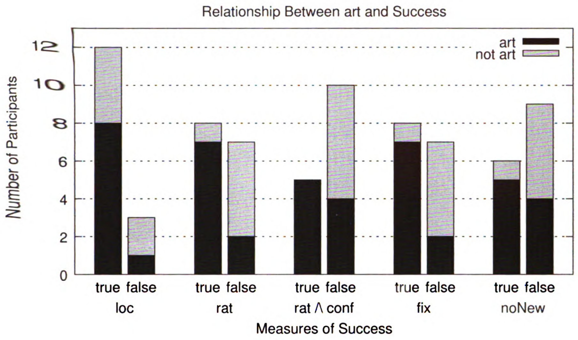


Figure 5.2: Relationships between failure-trace modeling and success attributes.



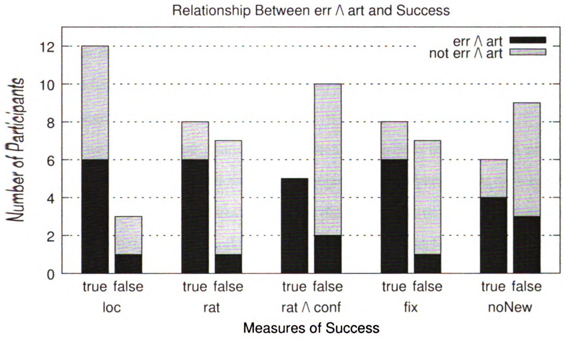
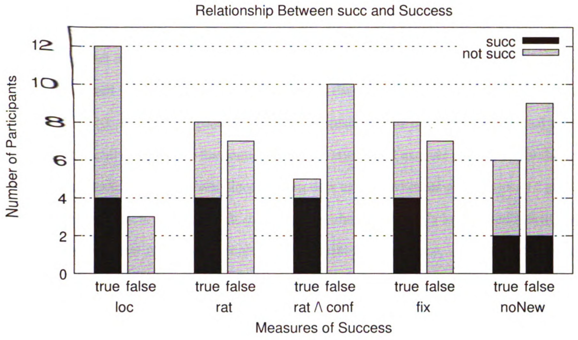


Figure 5.2 (cont'd)



‘ Figure 5.2 (cont’d)

explanation for this is that, although failure-trace modeling enabled participants to understand how the failure occurred, some modelers may still be lacking the design knowledge needed to change the code correctly.

Although we found a clear relationship between failure-trace modeling and success, we also found evidence of limitations with the approach. Two participants who created strongly-articulated models were unsuccessful at diagnosing the defect. Moreover, only four of the nine participants who produced strongly-articulated models produced the correct failure trace. This finding seems to be important because of the strong relationship we found between producing the correct trace and successfully diagnosing the defect with confidence.

We also checked to see if participants represented models externally, and if so, whether the models were strongly articulated. In this case, we evaluate the strength of articulation strictly in terms of the external representation, not taking into account any utterances made by the participant. None of the participants used the computer to represent a model. Four participants represented models on paper. Only participant 15 drew sequence diagrams to model interactions.[1] He drew three such diagrams, only one of which was both well-formed and strongly articulated. None of his diagrams ended in an error. Three other participants (10, 11, and 14) produced textual representations of models. A textual representation typically comprised the sequence of operations that execute during a hypothetical interaction. Participant 10 externalized two models as text, whereas participants 11 and 14 externalized one each. Of these four models, only one was strongly articulated (participant 10's); however, that model represented an infeasible failure trace. Overall, participants created very few external representations of models, and the models they did externalize tended to be of low quality.

5.1.3 Breadth-First Approach to Diagnosis

Table 5.4 depicts the events from the *root* and M subtree that each participant attempted to analyze. The bottom row indicates the percentage of events that participants discovered. The far right column provides the number of participants who analyzed each node.

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 (e.g., M_5–M_9). Every participant analyzed *root*, L_1, and M_1. Within M_1, thirteen participants analyzed M_2, and only three analyzed M_3. Within M_2, twelve analyzed M_4, and only two analyzed the M_5/M_6 subtree. Within M_3, no one analyzed M_7, and only one participant analyzed the M_8/M_9 subtree. Coverage is similar within the L subtree. Thirteen of the fifteen participants analyzed R_1. Twelve of the thirteen analyzed R_2. No participant analyzed every event in fault tree, and only participant 01 analyzed at least half of the events.

Recall that to be coded as bread, a participant must have analyzed a *sufficiently large*

[1] Another participant (06) drew a small sequence diagram that only depicted the actions of a single thread (not an interaction between multiple threads).

Table 5.4: Fault tree coverage by participants.

Node	Depth	Coverage by participant															Sum
		01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
root	0	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	15
L1	1	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	15
M1	1	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	15
R1	1	+	-	+	+	+	+	-	+	+	+	+	+	+	+	+	13
L2	2	+	-	+	-	-	+	+	+	+	+	+	+	+	+	+	12
L3	2	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1
M2	2	+	-	+	+	-	+	+	+	+	+	+	+	+	+	+	13
M3	2	+	+	-	+	-	-	-	-	-	-	-	-	-	-	-	3
R2	2	+	-	-	+	+	+	-	+	+	+	+	+	+	+	+	12
L4	3	+	-	-	-	-	+	+	+	+	+	+	+	+	-	+	10
L5	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
L6	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
L7	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
L8	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
L9	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
M4	3	+	-	-	+	-	+	+	+	+	+	+	+	+	+	+	12
M5	3	-	-	-	+	-	-	-	-	-	-	-	-	-	+	-	2
M6	3	-	-	-	+	-	-	-	-	-	-	-	-	-	+	-	2
M7	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
M8	3	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	1
M9	3	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	1
Pct. coverage:		52	29	29	48	24	43	33	43	43	43	43	43	43	48	43	

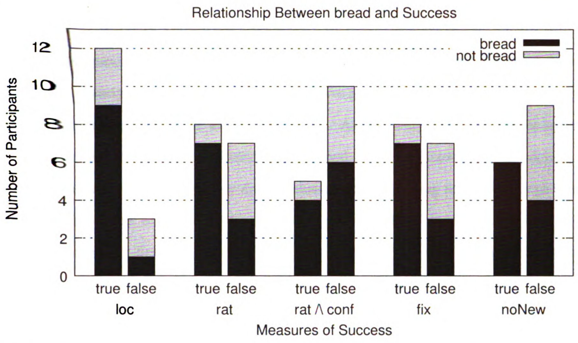


Figure 5.3: Relationships between use of the breadth-first approach and success attributes.

number of events in the fault tree down to depth two. In our study, a participant met this criteria if he analyzed events *root*, L_1, L_2, M_1, M_2, R_1, and R_2. Although events L_3 and M_3 were at depth two, we excluded them as outliers.[2] Table 5.2 lists the ten participants who we coded as bread. Only participant 01 analyzed all events down to depth 2 in the fault tree.

Our cross-case analysis did not find a significant relationship between the use of a breadth-first approach and success. Figure 5.3 depicts the frequencies of our cross comparison. Breadth-first participants were more successful than depth-first for all measures of success; however, none of the differences rose to the level of statistical significance. Seven breadth-first participants were successful at both diagnosing (rat) and fixing (fix) the defect (rat), whereas three were not successful by either measure. Only one depth-first participant was successful by these measures, whereas the other four were unsuccessful.

Focusing on the group we coded as having applied the breadth-first approach, nine

[2] Only one participant analyzed L_3, and only three analyzed M_3.

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 defects 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 problem-solving approach, (2) at some point during the study they abandoned this approach, 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 problem-solving approach, which should otherwise have led to the discovery of these events.

To see whether concurrency was a factor in this breakdown of the approach, we distinguish between a type of hypothesis whose analysis requires reasoning about *concurrent behavior* versus a type that only requires reasoning about *sequential behavior*. A hypothesis requires reasoning about concurrent behavior if its associated node in the fault tree has at least one of the following two properties. First, the node has a timing constraint that orders events executed by different threads. Second, the node is part of a conjunct of events (i.e., child nodes joined by an AND gate) and one of the events in the conjunct has a timing constraint that orders events executed by different threads. A hypothesis only requires reasoning about sequential behavior if its associated node in the fault tree has neither of these properties. The nodes in Table 5.4 that only require reasoning about sequential behavior are *root*, L_1–L_3, M_1–M_3, and R_1–R_2. The remaining nodes 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, five of the six events were analyzed by two or fewer participants, and only one (M_4) was widely analyzed. In the sequential category, three of the four nodes were analyzed by thirteen or more participants, and only one (M_3) was, for the most part, not analyzed. 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 problem-solving approach.

5.1.4 Systematic Comprehension

Our cross-case analysis did not find a significant relationship between the use of a systematic comprehension strategy and success. Ten of the fifteen participants used the strategy. Figure 5.4 depicts the frequencies of our cross comparisons. These data show that participants who applied the systematic strategy were, on average, more successful on task by all measures of success. However, this result is not statistically significant for any measure of success. 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.

Static and Causal Knowledge

We evaluated participants' static and causal knowledge with our poststudy questionnaire. We partitioned the questions into those that emphasize static knowledge and those that emphasize causal knowledge. Table 5.5 summarizes the participants' scores on the postquestionnaire. Participants generally did well on the static questions with a mean score of 84%. In contrast, they struggled on the causal questions with a mean score of only 55%.

To test the claim that the systematic strategy leads to a stronger mental model than the as-needed strategy, we compared the scores of the participants who used the systematic strategy (syst) with those who did not ($\neg$syst). Table 5.6 summarizes the results of the com-

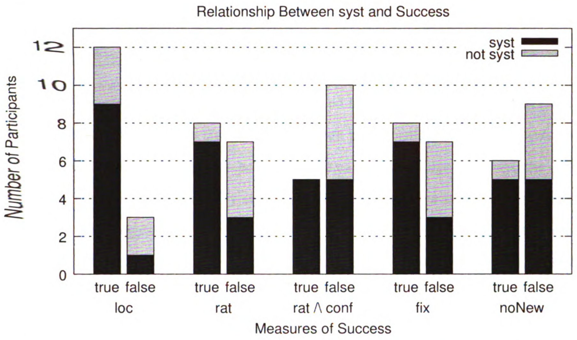


Figure 5.4: Relationships between use of the systematic comprehension strategy and success attributes.

Table 5.5: Poststudy questionnaire results.

Questions	Mean	Std. Dev.	Quartile				
			Min.	Lower	Med.	Upper	Max.
Overall	0.68	0.20	0.25	0.60	0.71	0.83	0.93
Static	0.84	0.19	0.41	0.73	0.89	1.00	1.00
Causal	0.55	0.23	0.13	0.47	0.53	0.74	0.88

Table 5.6: Relationship between systematic strategy and program knowledge.

Questions	syst		¬syst		Diff. in Means	Prob. (p)
	Mean	Std. Dev.	Mean	Std. Dev.		
All	0.73	0.21	0.57	0.16	0.16	0.13
Static	0.88	0.19	0.76	0.17	0.11	0.27
Causal	0.62	0.23	0.43	0.18	0.19	0.11

parison. We performed a heteroscedastic, two-tailed t-test to determine if the difference in means between the groups is significant. Although participants who used the systematic strategy exhibited higher scores on both static and causal knowledge questions, the difference in scores was not statistically significant. The absence of a statistically significant difference here suggests that the benefits of the systematic strategy with respect to a strong mental model may be reduced in the context of concurrent software. One possible explanation for this is that programmers who engage in the systematic strategy are implicitly analyzing data and control dependences. However, concurrent software makes reasoning about such dependences difficult, thus reducing the effectiveness of the systematic strategy. Section 2.1.2 provides rationale for this difficulty.

To test the claim that there is a relationship between a strong mental model and success on task, we performed a cross-case analysis of participants various levels of success. Table 5.7 depicts the results of our analysis. For each success attribute, we compared the scores of the successful and unsuccessful participants. We performed a heteroscedastic, one-tailed t-test to determine if the difference in scores between the groups is significant.[3] The results show significant differences in the causal knowledge of successful and unsuccessful participants for measures of success, except the weakest (loc). However, they only show a significant difference in static knowledge for noNew. This is consistent with Littman colleagues' claim that causal knowledge tends to distinguish successful and unsuccessful programmers.

[3] We use a one-tailed test here because being unsuccessful on task should not have a positive effect on postquestionnaire score. If we observed such a result, we would attribute it to random chance.

Table 5.7: Relationship between knowledge and success.

Attribute	Questions	Successful		Unsuccessful		Diff. in Means	Prob. (p)
		Mean	Std. Dev.	Mean	Std. Dev.		
loc	Overall	0.7	0.22	0.61	0.13	0.09	0.19
	Static	0.84	0.2	0.83	0.18	0.02	0.45
	Causal	0.59	0.25	0.44	0.1	0.15	0.07
rat	Overall	0.79	0.14	0.56	0.2	0.23	**0.02**
	Static	0.92	0.12	0.75	0.22	0.16	0.06
	Causal	0.69	0.17	0.41	0.2	0.28	**0.01**
rat $\wedge$ conf	Overall	0.79	0.16	0.62	0.21	0.17	0.052
	Static	0.92	0.15	0.8	0.2	0.12	0.11
	Causal	0.7	0.18	0.49	0.22	0.22	**0.04**
fix	Overall	0.79	0.14	0.56	0.2	0.23	**0.02**
	Static	0.92	0.12	0.75	0.22	0.16	0.06
	Causal	0.69	0.17	0.41	0.2	0.28	**0.01**
noNew	Overall	0.81	0.13	0.6	0.21	0.21	**0.01**
	Static	0.95	0.06	0.77	0.21	0.18	**0.02**
	Causal	0.7	0.17	0.46	0.22	0.24	**0.02**

Table 5.8: Performance on the poststudy questionnaire.

Type	Question	Overall	rat	rat ∧ conf	fix	noNew
Static	1	82	88/76	87/80	88/76	89/78
	2	80	88/71	**100/70 ***	88/71	83/78
	3	77	84/68	75/78	84/68	**96/64 ***
	4	83	**100/64 ***	**100/75 ***	**100/64 ***	**100/72 ***
	5.1	85	**100/68 ***	**100/78 ***	**100/68 ***	**100/75 ***
	6	88	**97/77 ***	95/84	**97/77 ***	**100/80 ****
	7	91	88/95	80/97	88/95	100/85
	11	83	94/71	90/80	94/71	**100/72 ***
	12	87	88/86	100/80	88/86	83/89
Causal	5.2	55	53/58	55/56	53/58	58/53
	8	63	67/57	61/63	67/57	73/55
	9.1	63	**78/45 ***	75/57	**78/45 ***	**83/49 ***
	9.2	52	60/43	60/48	60/43	63/44
	9.3	58	66/50	65/55	66/50	71/50
	10.1	68	70/66	72/66	70/66	71/66
	10.2	58	80/56	82/63	80/56	76/64
	13	33	25/43	40/30	25/43	17/44
	14	67	**100/29 ****	**100/50 *****	**100/29 ****	**100/44 ****
	15	44	**66/20 ***	65/34	**66/20 ***	**67/29 ***
	16	34	**55/11 ***	**68/18 ***	**55/11 ***	57/19
	17	60	**100/14 *****	**100/40 *****	**100/14 *****	**100/33 ****

Additionally, we performed a cross-case analysis of participants that achieved the four highest levels of success (rat, rat ∧ conf, fix, and noNew) and their scores on individual questions from the poststudy questionnaire. Table 5.8 summarizes performance on the individual postquestionnaire questions, listing the overall average, and for each category of success, listing the average for the successful group and the unsuccessful group (S/U). Question are sorted by the type of knowledge they are designed to evaluate (i.e., static or causal). For each question, we performed a heteroscedastic, one-tailed 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 (question 1). There were no significant differences between successful and unsuccessful participants on this question.

They also performed well on questions that evaluated knowledge of shared data, addressing shared objects and data structures (questions 4 and 5.1), scoring over 80% when asked to identify server classes that might be concurrently accessed by multiple threads (question 4) and when asked to describe the purposes of the main data structures (question 5.1). Significant differences were found on all categorizations of success for these questions. 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 (question 5.2) 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 (question 2), to state how many threads might play each role (question 3), and to answer questions about thread creation (questions 11 and 12). Performance on these questions was related to 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 (question 6). 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 three of the four categories.

Questions 7 and 8 addressed the behavior of the `dispatch_request` method, asking participants to list the major activities performed during this method (question 7) and to identify those activities in which the actor might block and explain the conditions and synchronization objects and operations involved (question 8). 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.

Questions addressing causal knowledge proved challenging. Two of these causal questions (questions 9 and 10) 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. We only observed a statistically significant difference for the noNew categorization, and only for question 9.1. Although participants performed well on these questions overall, a subgroup of scenarios proved problematic. 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 many actions 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 defect, asking participants to identify the nature of the synchronization defect that was seeded into the program. Question 14 was a multiple-choice question, in which the correct description of the defect was selected by 67% of the participants. Question 15 asked participants to describe, in their own words, the design defect that leads to the intermittent failure of 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 defect from a series of choices or than actually fixing the defect in the implementation.

Participants struggled when asked to consider interactions of the eBizSim server with an outside agent, the stress tester, and to discuss the effect of stress-tester speed in causing the server to crash (question 16). We found significant differences between the successful and unsuccessful groups under the rat, rat $\wedge$ conf, and fix categories.

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 (question 17). Interestingly, participants performed better (60%) at describing the fix than at describing the defect. We found a significant difference between the successful and unsuccessful groups under all categorizations.

In summary, participants were generally able to acquire static knowledge relevant to a multithreaded program, performing well at understanding key data structures and thread roles involved in synchronization and describing the life cycles 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 and colleagues' claim of a relationship between a strong mental model and success on task appears to hold here. However, the systematic strategy does not seem to be as effective at providing that model when applied to concurrent software. 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. We found that certain types of causal knowledge have a strong relationship with success, including the ability to describe concrete scenarios under which a defect might occur.

5.1.5 Cyclic Debugging

In our cross-case analysis, we found no relationship between the use of cyclic debugging and success. Thirteen of the fifteen participants instrumented the code (inst), and eleven refined their instrumentation based on a prior run of the program (refine). No participants succeeded in producing a failure trace with the technique. We partitioned the participants by the various attributes of success, but no partitioning showed a significant difference with respect to the use of execution-based tracing (inst or refine). Successful and unsuccessful participants at all levels employed the strategy. However, those in the group that correctly diagnosed the defect with confidence (rat $\wedge$ conf) appeared to use such tracing to localize the error to the `retrieve_request` method, and then to switch to other strategies. For example, one such participant ran the program once to confirm that the failure was produced, and then spent some time reading the code and locating the `dispatch_request` method. He then instrumented the code in that method to determine which of the several steps in that method was failing. While the program was running, he continued to study the code, but was "hesitant to do too much" until he "narrowed things down by at least one level." Once he achieved that, he sketched out a sequence diagram. However, he continued to run the program in the background, saying, "We'll just keep this going while we think." In contrast, those in the unsuccessful group appeared to continue to pursue cyclic debugging further.

5.1.6 Pattern Matching

Our cross-case analysis found evidence of a relationship between pattern matching and various levels of success. Only six of the fifteen participants recognized the wait-in-while idiom (pat). Figure 5.5 depicts the frequencies of our cross-case analysis. Participants who successfully diagnosed the defect (rat), did so with confidence (rat $\wedge$ conf, and fixed the defect (fix) all tended to do pattern matching significantly more than their unsuccessful counterparts ($p < 0.01$ in all cases). However, we did not find a significant relationship

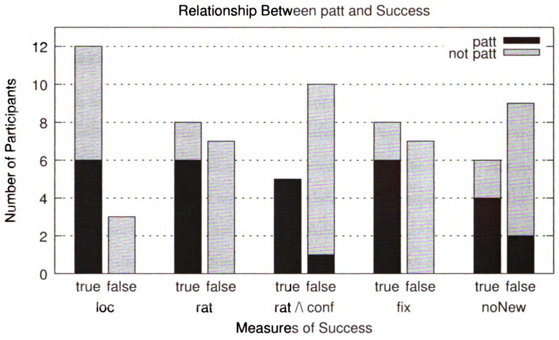


Figure 5.5: Relationships between recognition of the wait-in-while idiom and success attributes.

between the use of pattern matching (pat) and successfully fixing the defect without introducing new defects (noNew). This finding may arise because the eBizSim application uses a variant of the wait-in-while pattern. The variant augments the idiom with counters to prevent unnecessary signaling. Among the participants who introduced a new defect while fixing the defect (fix $\wedge \neg$noNew), the defects they introduced always involved misuses of these counters. Furthermore, we observed that all participants who recognized this violation of the wait-in-while pattern did so in the context of modeling. No participant merely recognized that the pattern had been violated and fixed the code solely on that basis.

The relationship between the recognition of the wait-in-while idiom and success suggests that training with such common patterns is another key element. That roughly half of the participants did not note this violation, despite a recent lecture on the topic in class, suggests that lecture alone may be insufficient to convey this concept and that hands-on exercises may be necessary. The limitations of pattern-matching should also be taught, so that programmers are mindful of when the technique is appropriate.

5.1.7 Tweaking the Code

Our cross-case analysis found evidence of a relationship between tweaking and failure at the task. Ten of the fifteen participants exhibited tweaking. Figure 5.6 depicts the frequencies of our cross comparisons. Participants who failed to explain the defect ($\neg$rat) and who failed to fix the defect ($\neg$fix) were significantly more likely to engage in tweaking (luck) ($p < 0.05$ in all cases). It seems likely that the relationship of the tweaking strategy with failure to correctly diagnose or correct the defect is a by-product of the failure to comprehend the nature of the defect; those participants did not know what else to do. The relationship between this strategy and an unsuccessful outcome suggests that programmers should not waste time in this way, and might better spend their time attempting to localize the error, looking for violations of well-established patterns, modeling the behavior of the system, and constructing candidate error traces.

5.2 Discussion

In summary, this exploratory study has provided insights into the strategies that programmers use to address the challenges of debugging concurrent software, and the practices programmers employ that are related to success on task. Our findings suggest that failure-trace modeling is an important factor in successful debugging. Participants who used the strategy were significantly more successful than those who did not. Moreover, greater commitment to the strategy yielded higher levels of success. For instance, we observed that participants who produced strongly-articulated error traces were more likely to correctly diagnose the defect with confidence. Furthermore, all participants who produced an actual failure trace diagnosed the defect with confidence and fixed the defect.

Although we found a clear relationship between failure-trace modeling and success, we

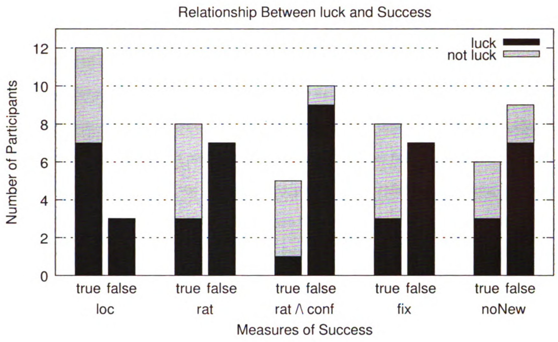


Figure 5.6: Relationships between code tweaking and success attributes.

also observed some limitations of the approach. Not all participant who produced strongly-articulated error traces were successful on task. Moreover, only a small number of failure-trace modelers were able to produce a model of an actual failure trace. The potential behaviors of concurrent programs are highly complex. The need to reason about such behaviors while performing failure-trace modeling may strain cognitive resources, such as working memory [7]. Such strain may hinder success with the strategy. This hypothesis is supported by the observation that participants modeled predominantly "in their heads," with only limited external support for reducing cognitive load.

We observed that the ability to recognize the violation of a particular concurrent programming idiom was closely related to success. Participants who noted our violation of the wait-in-while idiom were significantly more successful at diagnosing and fixing the defect than those who did not. This suggests the importance of bug smells [49] for generating hypotheses as to the cause of a failure.

Due to the nature of concurrency, programmers engaged in debugging must consider many plausible hypotheses as to the cause of the defect. These hypotheses arise from the multitude of possible behaviors a concurrent program may exhibit. The importance of managing hypotheses, so that they are investigated in a systematic fashion and are not lost or forgotten, is clear. However, we consistently found a breakdown in the breadth-first approach used by many of our participants. These participants failed to analyze many plausible hypotheses. Moreover, we found evidence that concurrency was a factor in whether or not hypotheses were analyzed. Hypotheses that required reasoning about interactions between multiple threads were analyzed far less often than those that required reasoning about the behavior of a single thread.

We observed several practices that were not related to success. Participants who tweaked the code, trying to fix the defect through luck, were significantly less successful than those who did not. A majority of participants engaged in cyclic debugging despite its well-understand limitations in the context of concurrent software. However, only the participants who also engaged in failure-trace modeling were successful on task.

The systematic comprehension strategy, which was previously found to lead to success on tasks [73], did not have a significant relationship with success here. One explanation is that the technique relies on the programmer performing an implicit dependence analysis as he engages in the strategy. With concurrent software, the dependences are difficult to understand, and this difficulty may account for the ineffectiveness of the strategy. One explanation given for the success of the systematic strategy is that it provides the programmer with causal knowledge about the program. Interestingly, we observed a significant relationship between the possession of causal knowledge and success on task. However, we did not find a significant relationship between the use of the systematic strategy and possession of such knowledge. This result suggests that successful participants gained causal knowledge through other means.

5.2.1 Limitations

We identified several aspects of our study that may limit how well our findings generalize to practice. The limited scale of the eBizSim application may be a limitation. This limitation was difficult to avoid because of the logistical constraints of the study. To address this problem, other types of studies, such as case studies, that allow for larger, more realistic programs are needed.

The composition of our participant pool may be a limitation. The behaviors of students may not be representative of programmers in general. Moreover, the CS students at Michigan State University that make up our participant pool are predominantly male, so our results may not generalize to female programmers. Continued sampling of different types of programmers is needed to address this limitation. In particular, industrial practitioners should be studied.

The absence of strong incentives for participants to succeed may be a limitation. Participants were only asked to participate for a relatively limited length of time. The natural incentives to succeed that exist in a real-world setting are difficult to duplicate in a study involving students. Again, other types of studies, such as case studies, with a more realistic structure of incentives are needed.

Our choice of seeded defect may be a limitation. The defect may not be representative of the kind of synchronization defects that arise in practice. More sampling of defects is required to address this limitation.

A potential limitation of the study concerns the use of the think-aloud because the cognitive resources required for introspection may affect how participants perform. Fortunately, numerous studies show that participants who only are asked to "verbalize their inner dialogue," as were the participants in this study, perform comparably on measures of performance with participants who are not asked to think aloud [38]. After an hour into our study, one participant who had thought that the method would be a hindrance stated, "[Talking aloud] turned out not to be a big deal. Especially while I was thinking through sequences of events, I pretty quickly became unaware of the fact that I was talking out loud."

CH

CO

EX

Our
tant
with
than
that
that
troll
abil
soni
to e

6.1

Our
mers
done
quenc

CHAPTER 6

CONTROLLED EXPERIMENT: PLANNING AND EXECUTION

Our exploratory study found that the use of the failure-trace modeling strategy is important for success when debugging concurrent software; however, it also found limitations with the strategy. Some participants who modeled were not successful on task, and less than half of those modelers were able to produce an actual failure trace. Our observation that participants predominantly modeled internally (Section 5.1.2) leads us to hypothesize that cognitive strain may have limited success. To test this hypothesis, we conducted a controlled experiment to test whether externally representing models improves a programmer's ability to reason correctly about the potential behaviors of concurrent software. Such reasoning is critical to success with failure-trace modeling; therefore if we observe a benefit to externalizing in this context, we can infer that the benefit should extend to the strategy.

6.1 Experiment Planning

Our experiment aims to test whether externally representing models improves programmers' reasoning ability in the aggregate. We compared external modeling with modeling done exclusively internally. The external representation took the form of multithreaded sequence diagrams (Section 2.4). Using a multithreaded program, a set of interactions, and a si

cor

6.1

Th

(CS

stu

mu

cou

wh

bac

abo

6.1

The

and

blo

serv

Pre

The

ithr

mak

a pa

have

of th

¹ W

a set of questions about the interactions, we measure reasoning ability as the number of correct answers.

6.1.1 Participants

The participants were 44 undergraduate CS students enrolled in a software design course (CSE 335) at Michigan State University. Students could volunteer to participate in the study, and were compensated with extra credit.[1] To enroll for the course, the students must have completed at least one course that emphasized programming in C++. One C++ course should provide adequate background to comprehend the programs in this study, which use only basic C++-language features. We did not expect participants to have a background in multithreaded programming. As part of the study, we taught participants about the multithreaded programming model and the ACE thread library.

6.1.2 Experimental Materials

The study materials comprise a *preexperiment questionnaire* (*prequestionnaire* for short) and an *experiment questionnaire* (*questionnaire* for short). We used the prequestionnaire to block and balance our treatment groups (see Section 6.1.4). The experiment questionnaire served as the primary study instrument.

Preexperiment Questionnaire

The prequestionnaire measures the ability to reason about the potential behaviors of a multithreaded program. To answer the questions correctly, the participant must understand the multithreaded programming model and be able to apply their knowledge to comprehend a particular program. We designed the prequestionnaire to measure how well participants have mastered the multithreaded programming model up to level 3 in Bloom's taxonomy of the cognitive domain [13]. This level emphasized the ability to apply a concept (e.g.,

[1] We offered alternative extra credit opportunities to students who did not want to participate.

a pro
gram
kept
minu
T
gran
mar
niza
eacl
nair
To
con
chr

thr
for
at
the
oc
tio
to
M
m
ei
pr

a programming model) in new situations (e.g., the comprehension of an unfamiliar program). The questions involved small, simple examples that required little context. We kept the scale small so that participants could complete the prequestionnaire in under thirty minutes.

The questions refer to a small (73 SLOC) multithreaded banking program. The program comprises four threads performing operations on a shared database object, which manages bank-account information. The database supports *readers-writer* style synchronization [64]—that is, it allows readers to concurrently execute its operations but provides each writer thread with exclusive access while executing operations. In the prequestionnaire program, two of the threads play the role of reader and two play the role of writer. To keep the program small, we elided much of the "business logic"—that is, the code concerned with the specific banking functions—so that participants could focus on the synchronization logic.

The prequestionnaire presents seven scenarios of interaction involving reader and writer threads. For each scenario, it asks one or two multiple-choice questions about the scenario for a total of eight questions. The questions either ask about the state of the program at the end of the scenario, about the behaviors that might happen immediately following the scenario, or about why certain events (e.g., a state transition or operation invocation) occurred during the scenario. Figure 6.1 depicts a scenario and question from the prequestionnaire. Appendix B provides a listing of the program source code that the question refers to. Observe that the scenario describes an interaction among a reader and a writer thread. Many details of the interaction are elided. To answer the question correctly, the participant must apply his knowledge of the semantics of the synchronization mechanisms to infer the elided actions and state transitions. Appendix B provides a complete reproduction of the prequestionnaire along with solutions to the questions.

Ex

Th

ab

pe

sn

si

se

a

re

req

gra

abo

sync

Scenario: An interaction involving one reader R and one writer W. Assume W is in the running state within an invocation of `start_write` and R is in the ready state. A context switch occurs just after W increments `n_writers` by one. W transitions to the ready state, and R transitions to the running state. R invokes `start_read` and quickly transitions to the blocked state.

Question: Why does R enter the blocked state? (select one of the following)

(a) R enters the blocked state via the call to `wait` on line 50 because `n_writers` is non-zero at the time.

(b) Deadlock occurs. Because W holds the lock, R can't possibly acquire the lock.

(c) R enters the blocked state via the call to `wait` on line 36 because `n_writers` is non-zero at the time.

(d) R enters the blocked state via the call to `acquire` on line 35.

(e) R enters the blocked state due to the occurrence of a context switch.

Figure 6.1: Sample preexperiment-questionnaire scenario and question.

Experiment Questionnaire

The experiment questionnaire, like the prequestionnaire, measures the ability to reason about the potential behaviors of a multithreaded program; however, the questions in the experiment questionnaire are designed to be more cognitively taxing. The questions refer to a small (56 SLOC) multithreaded server program, which we seeded with a defect. The server simulates an e-business server that accepts and processes requests from remote clients. The server comprises multiple threads, each of which plays one of two distinct roles—that of a *listener* or a *handler*. 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 processing the requests. We seeded the program with a defect because we are primarily concerned with how programmers reason about thread interactions in the context of debugging tasks. The defect is related to thread synchronization and allows a handler to erroneously invoke the pull operation on an empty request queue.

The questionnaire presents four scenarios with one question each. The questions are all the same. They ask whether the scenario is consistent with the source code, and if so, whether the scenario causes the program to enter a clear error state. Figure 6.2 depicts an example of a scenario and question. Appendix B provides a listing of the program sources that the scenario refers to. Observe that the scenario describes an interaction among a listener thread and two handler threads. As with the prequestionnaire, many details of the interaction are elided and must be inferred by the participant. Appendix B provides a complete reproduction of the questionnaire along with solutions to the questions.

6.1.3 Hypotheses, Parameters, and Variables

Our experiment has one independent variable: the use of external representations of thread interactions. It has two treatments: the exclusive use of internal representations (*internal*) and the use of external representations in the form of sequence diagrams (*external*). It has one dependent variable: the ability to reason correctly about thread interactions (C). We measured C using the experiment questionnaire and report the measurement as the proportion of correct answers (i.e., a score in the range $[0.0, 1.0]$).

Table 6.1 lists our hypotheses for testing the effect of using external representations. The null hypothesis (H_0) states that using external representations has no effect on ability to reason—that is, there is no difference in the mean scores of questionnaires filled out while externalizing and those filled out using only internal representations. The alternative form of the hypothesis (H_a)—that is, what we expect to happen—states that externalizing yields better scores on the questionnaire than using only internal representations. We do not anticipate the result $C(external) < C(internal)$ because the participants have access to their internal facilities while externalizing. If we observed such a result, we would assume it was due to random chance or a defect in the experiment (e.g., participants were not given enough time to complete the questionnaire).

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 6.2: Sample experiment-questionnaire scenario and question.

Table 6.1: Hypotheses tested by our experiment.

Dependent variable	Null hypothesis (H_0)	Alternative hypothesis (H_a)
Score on questionnaire	$C(external) = C(internal)$	$C(external) > C(internal)$

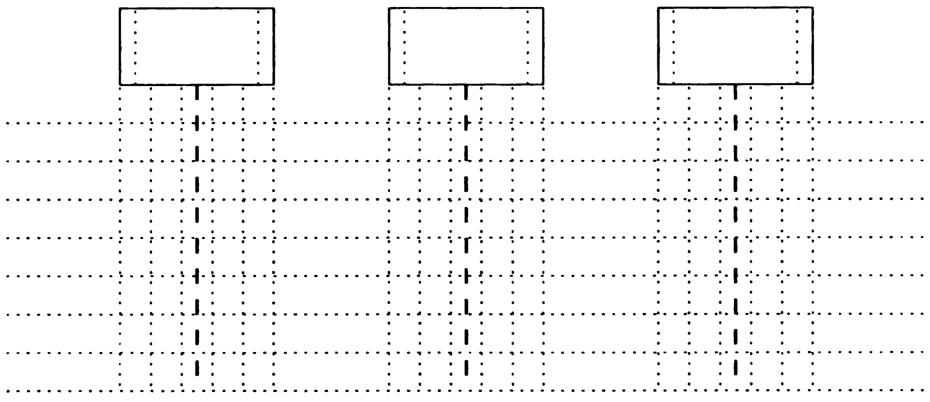


Figure 6.3: Part of a sequence-diagram template.

6.1.4 Experiment Design

Our experiment had one factor (i.e., whether models of thread interactions are externalized as sequence diagrams) and two treatments (i.e., externally representing models as sequence diagram and internally representing models). We used a *between-subjects* design—that is, each participant received only one of the treatments. We partitioned our participants into two treatment groups: the *external group* and the *internal group*. We used the preexperiment questionnaire to balance the groups. To do the partitioning, we ordered the participants by their prequestionnaire scores. We randomly ordered participants with the same score. We assigned alternating participants to each group. This approach produces groups with similar mean scores and standard deviations.

To test our hypothesis, the external group and the internal group both filled out the experiment questionnaire. We asked the external group to draw a sequence diagram of each scenario before answering questions about that scenario. We provided them paper printed with sequence-diagram templates (Figure 6.3) to help them draw the diagrams. In contrast, we the asked the internal group to answer the questions "in their heads"—that is, without drawing pictures or writing notes. We collected all testing materials from participants, so that in addition to evaluating their answers, we could analyze their diagrams.

6.2 Execution

6.2.1 Preparation

Prior to running the study, we recruited participants from the software-design course. We sent an email to the enrolled students soliciting participation. We used extra credit in the course as an inducement. We offered an alternative extra-credit homework assignment, so students who opted not to participate could also earn the extra credit. Students indicated their participation by filling out a consent form. The form described the activities involved with participation as well as the time commitment. The form reassured participants that any materials they produced for the study (e.g., filled-out questionnaires) will either be kept private or, if shared, be anonymized using randomly-assigned ID numbers. To help ensure an acceptable level of motivation from participants, we deceived participants by stating that participants who perform poorly on the study questionnaires will not receive the extra credit. Deception is commonly employed in human-subjects studies and is deemed ethical if there is no risk of harm to the participants [3]. We assured the participants that if they made a reasonable effort they would receive the credit. In truth, we awarded all participants the extra credit, regardless of how they performed on the questionnaires. We collected 49 completed consent forms.

6.2.2 Execution of Experiment Procedure

The entire study involved three 80-minute sessions spread out over two weeks. All sessions were held during the course's regular meeting time. Non-participants who were enrolled in the course were permitted to attend the sessions. During the first two sessions, participants received a two-part lecture on multithreaded programming in C++. The parts were 80 and 50 minutes long, respectively. As part of the lecture, we taught students how to use our multithreaded sequence-diagram extension (see Section 2.4.2). During the last 30 minutes of the second session, participants filled out the preexperiment questionnaire. We collected

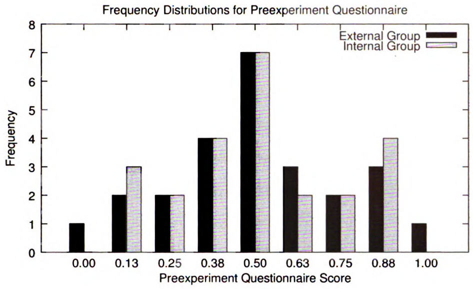


Figure 6.4: The frequency distributions of the scores of the external and internal groups on the preexperiment questionnaire.

50 completed preexperiment questionnaires.

Between the second and third sessions, we graded the prequestionnaires and partitioned the participants into treatment groups. We used the procedure described in Section 6.1.4 to perform the partitioning. After partitioning, the external and internal groups had similar means and standard deviations:

External Group:	Mean = 0.510	Std. Dev. = 0.255
Internal Group:	Mean = 0.505	Std. Dev. = 0.240

Figures 6.4 and 6.5 provide some additional descriptive statistics. The statistics support that our partitioning produced well-balanced groups with respect to the prequestionnaire.

For the final session, the treatment groups met in separate rooms and filled out the experiment questionnaire. We chose to have them meet in separate rooms, so that if one group tended to finish faster than the other, the slower group would not feel pressure to rush. We collected 44 completed questionnaires along with any drawings and notes that participants produced. Finally, we graded the experiment questionnaires and performed our analysis.

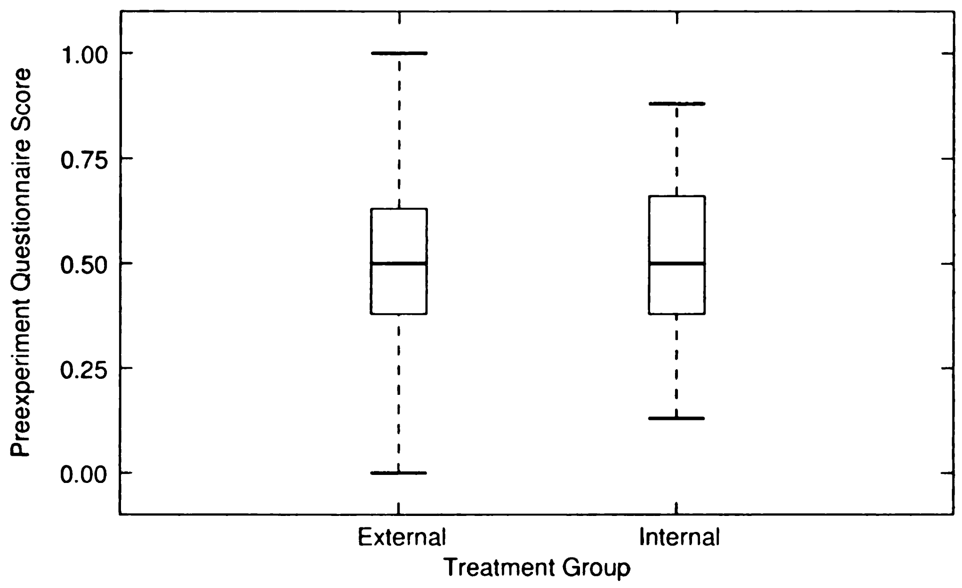


Figure 6.5: The scores of the external and internal groups on the preexperiment questionnaire.

6.2.3 Data Validation

We did not throw out any experiment questionnaires. However, six participants who completed the prequestionnaire did not complete the experiment questionnaire. One of the six never filled out a consent form and presumably was not a participant in the study. The other five dropped out of the study prior to taking the experiment questionnaire. The attrition had only a minor impact on the balancing of the treatment groups. The final groups mean and standard deviations on the prequestionnaire were as follows:

External Group:	Mean = 0.494	Std. Dev. = 0.260
Internal Group:	Mean = 0.494	Std. Dev. = 0.236

CHAPTER 7

CONTROLLED EXPERIMENT: ANALYSIS AND DISCUSSION

Having administered and scored the experiment questionnaire, we performed a quantitative analysis, which compares the scores of the treatment groups to see if their was a statistically significant difference. Additionally, we perform two supplementary qualitative analyses. Section 7.1 details our analyses and the results that emerged from them. Section 7.2 discusses our interpretation of the analysis results and threats to the validity of the experiment.

7.1 Analysis

In this section, we present descriptive statistics for the data that we collected in the experiment. Then, we address our hypothesis that tests the effect of externalizing models of thread interactions with sequence diagrams on the ability to reason correctly about potential program behaviors. Finally, we present some supplemental analyses of the data.

7.1.1 Descriptive Statistics

Table 7.1 provides statistics regarding the results of the experiment questionnaire. The table shows the results for the questions individually as well as in the aggregate. For each of the

Table 7.1: Results of the experiment questionnaire.

Question(s)	External Group		Internal Group		Diff. in Means	Prob. (p)
	Mean	Std. Dev.	Mean	Std. Dev.		
1	0.59	0.50	0.32	0.48	0.27	**0.04**
2	0.64	0.49	0.14	0.35	0.50	**0.0002**
3	0.55	0.51	0.45	0.51	0.09	0.28
4	0.41	0.50	0.32	0.48	0.09	0.27
1–4	0.55	0.31	0.31	0.27	0.24	**0.005**

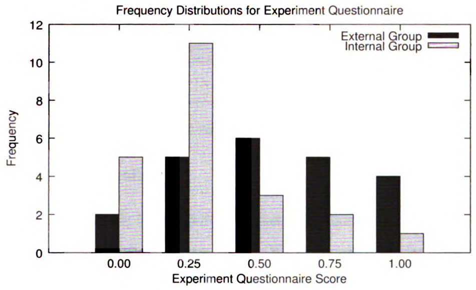


Figure 7.1: Frequency distribution for the experiment questionnaire scores.

four questions, we scored participants one point if they answered correctly; otherwise, we scored zero points. We computed each participant's overall score as the mean of their scores on the questions (i.e., overall score in the range $[0.0, 1.0]$). The results show that the external group produced a higher mean score than the internal group on all questions. Figures 7.1 and 7.2 provide additional descriptive statistics.

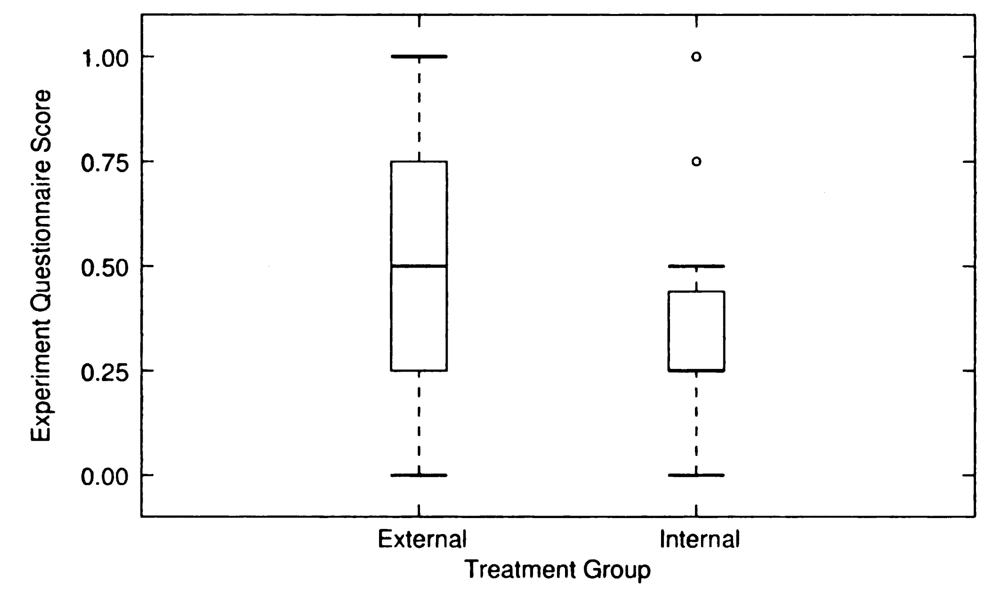


Figure 7.2: Box plot of the experiment questionnaire scores.

7.1.2 Hypothesis Testing

We performed a two-sample t-test to assess the statistical significance of the difference in the questionnaire scores of the treatment groups. We performed a similar t-test on the individual questions. We used a one-tailed t-test because (1) we predicted that the external group will 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 (i.e., no significant difference) regardless of the size of the difference. We set the level of significance (α) to 0.05. We report the results (i.e., the p values) of the t-tests in the rightmost column of Table 7.1. The difference in means is greatest on questions 1 and 2, and we found both to be statistically significant. The differences in means for questions 3 and 4 is smaller, and is not statistically significant. Looking at the results for all the questions in the aggregate, the external group scored 0.27 higher than the internal group, and the difference is statistically significant. Based on the difference in overall scores, we rejected our null hypothesis H_0 and accepted our alternative hypothesis H_a.

7.1.3 Supplementary Analyses

To supplement our hypothesis-checking analysis, we performed two additional analyses of the data. Recall that each question on the experimental questionnaire has a scenario that describes a thread interaction. We analyzed the complexity of the interactions to see if there was a relationship with the differences in scores of the two treatment groups. We also analyzed the quality of diagrams produced by the external group to see if there was a relationship with performance on the questionnaire.

Interaction-Complexity Analysis

Using the approach from Section 2.5, we applied several metrics to the four scenarios of interaction from the questionnaire. We first modeled the program from the questionnaire as an LTS in the FSP language [76]. Appendix D provides a listing of our FSP code. The resulting model has 1097 states and 2416 transitions. Next, we computed a trace that represents each interaction using the LTSA tool. From each trace, we computed the total number of transitions, the number of threads involved, the number of context switches, and the number of block/unblock actions. Appendix D provides a complete description of the traces. We report these measurements in Table 7.2. The rightmost column of the table lists the differences in the external and internal group scores on the questions (statistically significant differences in bold). Based on the data, there is no apparent relationship between the number of transitions or the number of threads and the difference in scores. However, the number of context switches and block/unblock actions are noticeably higher for the questions that showed larger differences in scores. This pattern suggests that as scenario complexity increases under these metrics, the effect of externalizing also increases.

Diagram-Quality Analysis

We performed an analysis to check for a relationship between diagram quality and questionnaire score. We evaluated the sequence diagrams collected from the external group.

Table 7.2: Complexity measurements over the thread interactions from the questionnaire.

Question	Transitions	Threads	Context Switches	Blocks/Unblocks	Diff. in Means
1	22	3	4	4	**0.27**
2	11	3	3	4	**0.50**
3	15	3	2	1	0.09
4	13	3	2	1	0.09

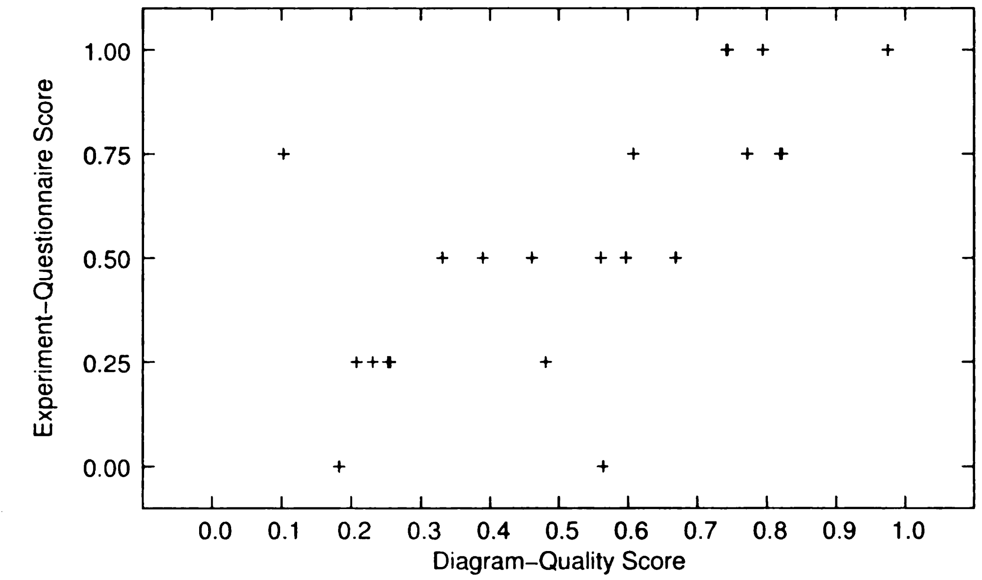


Figure 7.3: Plot of the relationship between diagram quality and questionnaire score.

We assigned a score to each diagram in the range [0.0,1.0]. The score captured not only the correctness of the 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. We did not require participants to use our sequence-diagram extension (see Section 2.4.2); however, participants who used the extension tended to omit fewer details. Appendix B provides a complete description of the evaluation rubric. We assigned each participant in the external group an overall diagram-quality score, which is the mean of their individual diagram scores. Figure 7.3 depicts the relationship between questionnaire and diagram scores. The scatter plot suggests that as diagram quality increases, the ability to reason about potential program behaviors also increases.

7.2 Discussion

In this section, we discuss the results of the experiment. First, we evaluate the results and discuss their implications for practice. Second, we discuss threats to the validity of the results.

7.2.1 Evaluation of Results and Implications

Overall, the results support our hypothesis that externally representing models of thread interactions with sequence diagrams improves ability to correctly reason about the behavior of concurrent programs. We were able to reject the null hypothesis because the external group performed significantly better on the experiment questionnaire than the internal group ($p < 0.05$). This finding supports our claim that externalizing improves the rate of success with the failure-trace modeling strategy.

Our results also suggest that interaction complexity is a key factor in reasoning. By analyzing complexity-relevant attributes of the interactions from the experiment questionnaire, we found that the effect of externalizing was greater for the interactions with higher levels of these attributes. In particular, the external group scored noticeably higher on interactions with more context switches and block/unblock actions. Questions with lower levels of these attributes exhibited a much smaller difference. This finding suggests that the cognitive strain associated with reasoning about complex interactions may be an important factor in the ability to reason correctly. We hypothesize that external representations alleviate the cognitive strain, thereby improving programmer's ability to reason correctly. These results also suggest that programmers can model internally up to some threshold of complexity without significant negative effects on reasoning ability.

Finally, our results suggest that high-quality diagrams are important for successful reasoning. Diagram quality comprises both the correctness of the diagram and the level of detail. By analyzing the sequence diagrams produced by the external group, we found

evidence that questionnaire scores tend to increase as diagram quality increases.

7.2.2 Threats to Validity

This section discusses threats to the validity of our experiment and how we addressed them. We adopt the categories and terminology prescribed by Wohlin and colleagues [131]. They recognize four types of validity: *conclusion*, *internal*, *construct*, and *external*.

Conclusion Validity

Conclusion validity is the degree to which correct conclusions can be drawn about relations between the treatment and the outcome. We recognize the following possible threats.

Reliability of measures: Questions on the questionnaires may be ambiguous or difficult for participants to understand. We addressed this threat by having multiple investigators review the questionnaires as well as conducting a pilot run of the experiment using the questionnaires.

Reliability of treatment implementation: The treatment groups filled out the experiment questionnaire in separate sessions. The administration of the sessions may have varied, thus affecting participants' outcomes. We addressed this threat by providing the same script to the investigators who administered the questionnaires. Moreover, we provide the investigators with the same instructions regarding how to address questions from participants.

Internal Validity

Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variable. We recognize the following possible threats.

Maturation: Participants may have improved their mastery of concurrent programming between the time that they completed the preexperiment questionnaire and the experiment questionnaire—in fact, simply filling out the preexperiment questionnaire may have caused them to learn. Such learning can throw the treatment groups out of balance if one group learns faster than the other. We minimized this threat by keeping the time between the preexperiment questionnaire and experiment questionnaire short (i.e., less than a week). Moreover, we ensured that neither group was given more motivation to learn than the other. For example, we did not reveal to participants the types of questions they would be asked to answer prior to filling out the experiment questionnaire.

Resentful demoralization: The internal group may have felt unduly restricted by using only internal representations of the thread interactions and underperformed on the experiment questionnaire. Similarly, the external group may have resented the additional work associated with drawing the sequence diagrams and underperformed. We addressed this threat by administering the experiment questionnaire to the groups simultaneously in separate rooms. Moreover, we did not reveal to each group the instructions that the other group received.

Construct Validity

Construct validity is the degree to which the independent and dependent variables accurately measure the concepts they purport to measure. We recognize the following threat.

Inadequate preoperational explication of constructs: Ability to reason correctly about thread interactions is a difficult concept to completely measure. Participants must be able reason about an interaction at some level to answer the questions in the experiment questionnaire. Moreover, someone who cannot answer those questions must be limited in their reasoning to some extent. However, there may be types of reasoning that our questions failed to test.

External Validity

External validity is the degree to which the results of the research can be generalized to the population under study and other research settings. We recognize the following possible threats.

Interaction of selection and treatment: The participants in our study may not be representative of professional software engineers. However, the results can be useful in an industrial context for two reasons [14]. First, studies performed in laboratory settings allow the investigation of a larger number of hypotheses at a lower cost than field studies. Second, the hypotheses that seem to be supported in the laboratory setting can be subsequently tested in more realistic industrial settings with a better chance of discovering important and interesting findings.

Interaction of setting and treatment: The programs and scenarios used in this study may not be representative of industrial software in terms of their size and complexity. We kept materials relatively small in scale to meet the time constraints of our controlled experiment. We partially addressed this threat by giving the e-business server a realistic multithreaded architecture based on the Reactor design pattern [107].

Several different application domains emphasize the use of concurrency (e.g., reactive systems and scientific applications). Programmers in these domains commonly employ different architectures and programming models. Our study emphasized reactive server applications written using the POSIX threads programming model. Therefore, the results of our study may not generalize to other application domains.

CHAPTER 8

DISCUSSION

The goal of our work is to understand the strategies and practices that successful programmers employ during the debugging of concurrent software. The empirical studies described in the previous chapters have contributed several key findings toward our goal and generated interesting new research questions. Additionally, we learned practical lessons in the conduct of empirical software engineering during the course of these studies. In this section, we discuss our research findings and their implications for practice (Section 8.1), the lessons learned (Section 8.2), and future work (Section 8.3).

8.1 Research Findings

An important finding of our work is that successful programmers use failure-trace modeling during debugging. Our exploratory study found that successful participants were significantly more likely to use the strategy than unsuccessful. However, we also observed some participants who were unsuccessful with the strategy. Two such participants were completely unsuccessful on task, and many more, although successful at fixing the defect, failed to produce an actual failure trace. Participants predominantly modeled internally, and cognitive overload may have been an important factor in these cases. External representations can improve performance on cognitively-taxing tasks [139]. Our controlled experiment found that creating external representations of models indeed improves programmers ability to reason correctly about potential program behavior. Participants in the experiment who drew sequence diagrams of thread interactions were significantly more success at reasoning about potential behavior than those who worked exclusively internally.

Our findings with respect to failure-trace modeling have important implications for practice. The benefits afforded by failure-trace modeling make the strategy a strong candidate for tool support. A failure-trace modeling tool might provide features for creating and manipulating sequence diagrams. A graphical interface could deliver an externalization similar to that used in our experiment. Having such a tool would implicitly encourage programmers to engage in failure-trace modeling. Currently available sequence-diagramming tools (e.g., Papyrus[1]) provide limited support for concurrency, but they could be adapted to support our multithreaded extension (Section 2.4).

Our experiment also found that correct and highly-detailed diagrams yield the greatest benefits. The failure-trace modeling tool could have features for detecting and preventing errors in diagrams. For instance, the tool could automatically analyze the buggy source code to generate a *call graph*—that is, a directed graph that represents calling relationships between the methods in a program [104]. The tool could use the graph to warn the programmer about potentially missing method calls in the diagram. Another common source of errors is the complex and subtle semantics of synchronization mechanisms [105]. The tool could automatically track the state of synchronization objects and enforce the proper semantics of their operations to prevent them from entering invalid states. In addition to ensuring correct diagrams, the tool could also encourage the programmer to add more details to their diagrams. The programmer could be prompted to add some of these details, and the tool could automatically elaborate others. For instance, the tool could automatically add hatching to an actor's activations when the programmer specifies the invocation of a synchronization operation (e.g., condition-variable `wait`) that results in blocking. The tool could also actively help the programmer produce a failure trace. Based on one diagram,

[1]`http://www.papyrusuml.org/`

it could automatically suggest alternative diagrams that exhibit different thread schedules. Sequence diagrams could also be dynamically generated from specific executions of the program [15, 16]. Even if the programmer is unable to reproduce the failure, a generated diagram of a non-failing run may still provide a useful starting place to begin searching for the defect.

Another important finding was that concurrency thwarts attempts to systematically manage hypotheses regarding the cause of the defect. Such management of hypotheses is known to be key to success in debugging [123]. The explosion of plausible hypotheses induced by concurrency makes this practice all the more important. Our exploratory study found many participants engaged in a breadth-first approach to diagnosing the defect. However, we observed a consistent breakdown in the approach, and as a result, many hypotheses were not investigated. We see two possible causes for this breakdown. First, participants may have been forgetting to check hypotheses. Although the buggy program was small, it engendered a comparatively large number of plausible hypotheses. Between the tasks of reasoning about the program's behavior and managing the hypotheses, participants may have been under a heavy cognitive load. As with failure-trace modeling, participants predominantly managed hypotheses internally. Thus, they may have been prone to forgetting hypotheses that they postponed checking. Second, participants may have had difficulty identifying some hypotheses. The complexity of concurrent software makes it difficult to reason about its potential behaviors. If the formulation of a particular hypothesis requires reasoning about a subtle interaction among multiple threads, then programmers may fail to recognize the hypothesis. Indeed, our data show that participants were much less likely to analyze hypotheses that involved reasoning about the behavior of multiple concurrently-interacting threads than hypotheses that involved reasoning about a single thread.

The importance and difficulty of managing hypotheses also makes the practice a prime candidate for tool support. Such a tool could enable programmers to externalize their hypotheses. One debugging expert [138] prescribes externalizing hypotheses as textual lists with accompanying notes regarding the state of each hypothesis (e.g., confirmed or refuted). Based on our experiences modeling the space of hypotheses with fault trees, we propose a tool that organizes hypotheses in a graphical fault tree, which incorporates our multithreaded extensions (Section 5.1.3). Our tool would record the same information as the textual approach as well as conveying the structural relationships between the hypotheses. We may be able to extend existing fault tree tools (e.g., OpenFTA[2]) for this purpose. These features focus on addressing the issue of remembering hypotheses. We can also imagine features to help programmers identify hypotheses. For example, a multithreaded-program slicer (e.g., Indus[3]) could be used to suggest potential dependences.[4] Dependence information is known to be useful in formulating hypotheses [138]. However, slicers have not been widely used for concurrent software for debugging, so it is an open question whether they are effective.

Another finding of our work was that the systematic comprehension strategy, which others [73] have claimed leads to success on programming tasks, does not lead to success when applied to concurrent software. We observed ten of our participants using the strategy; however, we did not find a significant relationship between its use and success. One of the claimed benefits of the strategy is a strong mental model, which comprises both static and causal knowledge of the software [73]. The systematic strategy is supposed to be especially effective for gathering causal knowledge. In contrast, the as-needed comprehension strategy is thought to be less effective for gathering this type of knowledge. However, we found that participants who obtained higher levels of knowledge (causal or otherwise) were not significantly more likely to use the systematic strategy than the as-needed strategy. The downfall of the strategy may be that it involves an implicit dependence analysis. That is, in tracing through the control flow of the program (as per the systematic strategy), the programmer gains knowledge of data and control dependences. Unfortunately, such

[2]`http://www.openfta.com/`

[3]`http://indus.projects.cis.ksu.edu/`

[4] Slicers tend to be imprecise, so it is up to the programmer to determine whether a reported dependence is an actual one.

dependences are difficult to identify in concurrent software.

As predicted by the literature [73], we found that the possession of a strong mental model, and especially causal knowledge, was related to success. This finding suggests that programmers acquire causal knowledge through means other than the systematic strategy. Failure-trace modeling may engender such causal knowledge. It stands to reason that by analyzing possible thread interactions, the programmer could discover subtle data and control dependences. However, unlike the systematic strategy, failure-trace modeling is not applied globally to the system. Therefore, failure-trace modeling generates causal knowledge in an as-needed fashion. The literature claims that the systematic strategy cannot be applied practically to large software, and successful programmers of such software use a *methodical* (albeit as-needed) approach to comprehension [100]. Like large software, the complexity of concurrent software also appears to necessitate as-needed comprehension. Therefore, a methodical approach may be associated with success in our context as well. Unfortunately, we were unclear how to code for the methodical approach in our study because the original analysis of the approach emphasized the navigation behavior of participants in the context of a large software system. In contrast, the application in our study was small and involved little navigation.

A surprising observation from our think-aloud study was that participants engaged heavily in cyclic debugging. Even those who demonstrated a capacity for strongly-articulated modeling exhibited this behavior. Almost all of the participants began by using the execution-based tracing strategy. However, successful participants tended to use the strategy only to localize the defect. They switched to failure-trace modeling to actually produce a failure trace. Many participants ran an instrumented program in the background throughout the session. Furthermore, many used the approach in an attempt to confirm (by absence of failure) the correctness of their solution. The participants who continued to invest in cyclic debugging long past the point of diminishing returns, instead of switching to failure-trace modeling, were ultimately unsuccessful. Many participants expressed understanding the limitations of the technique, but continued to use it anyway. This suggests that more hands-on training is needed to develop greater "buy in" regarding the limitations of the cyclic debugging. Such training could involve laboratory exercises using artifacts similar to those in our study. Thus, students would gain first-hand experience with these limitations.

8.2 Lessons Learned

During large, complex studies, such as ours, many practical issues arise. We gathered insights into how to address some of these issues throughout the course of our work. In this section, we discuss two particular issues we encountered and our experiences and insights in addressing them. The first issue pertains to the difficulty of packaging think-aloud data of programmers into a form that is usable to other researchers and economical to produce. The second issue pertains to the difficulty of transcribing many hours of think-aloud sessions.

8.2.1 Sharing Think-Aloud Data

In reporting on a think-aloud study, sufficient think-aloud data must be provided so that other researchers can replicate the analysis or run their own analyses [89]; however, we found satisfying this requirement problematic. Traditionally, textual transcripts of participants' utterances are shared with other researchers. Such transcripts are sufficient when participants use predominantly internal representations while engaged in tasks. However, we found such transcripts insufficient for our study.

The programmers in our study worked with a rich set of tools and documents throughout their tasks. To interpret their utterances correctly requires a detailed understanding of the context in which each utterance was made. Examples of such context include the source file currently in view and whether the participant is passively reading or actively editing a file. Unfortunately, properly documenting the context is difficult. Recording detailed *action protocols*, which are transcripts of the observable actions taken by participants, is not practical because the volume of information that must be encoded is overwhelming. For example, consider the amount of text needed to describe the activity of a mouse pointer over a two-hour session. Attempts to reduce the level of detail by abstracting the actions produced highly subjective protocols, which would make it difficult for other researcher to validate our work. Moreover, we may omit details that other researchers need to test their own theories.

To address these problems, we propose distributing think-aloud data of programmers as subtitled, silent screen-capture videos. Protecting the identities of participants is a foremost ethical concern in human-subjects studies [3]. Therefore, sharing raw videos (with audio intact) is unacceptable. Identifying characteristics such as a participant's voice or likeness should not be released under any circumstances. Video editing software can be used to remove the audio track, which effectively *sanitizes* (i.e., removes sensitive information from) the data. Of course, participants must be instructed not to do anything during their session, such as checking email, that would give away their identity. However, in the event that such a slip occurred, video-editing software could be used to remove or block out the offending parts of the video.

Subtitling the videos effectively synchronizes a verbal transcript with the detailed context information that is needed for interpretation. Software applications for editing subtitles are widely available (e.g., Jubler[5]). Conveniently, they commonly store subtitles in a text file that is separate from the video file. Video-player applications (e.g., Window Media Player) automatically insert the subtitles into the video during playback. Because the subtitle file is in plain text, it is not difficult to translate the file into another format, such as a comma-separated volume (i.e., a spreadsheet). Thus, a subtitle editor can be used to simultaneously subtitle the video and produce a transcript of the utterances.

In addition to allowing others to validate your work, these subtitled videos provide an

[5]`http://www.jubler.org/`

excellent resource for other researchers. Collecting and transcribing think-aloud data is costly and labor intensive. This is especially true in the case of programmers of concurrent software where participants with the necessary skill set are difficult to find and often need some additional training (as was the case in our study). Videos could be used by other researcher to economically develop and test new theories. Moreover, the videos could be used to test and refine new study designs before "going live."

8.2.2 Transcribing Think-Aloud Sessions

Transcribing think-aloud sessions is tedious and time consuming. In an effort to ease the task, we employed a software application, Dragon NaturallySpeaking[6], which is reputed to be among the best transcription tools. The software provides functionality for translating spoken words recorded in an audio file (e.g., MP3) to text. It is recommended that the software be "trained" for the person's voice to be transcribed. Each participant engaged in a thirty-minute training session with the transcription software prior to their think-aloud session. Unfortunately, despite this training, the transcripts produced by the software were completely unusable. Although the software produced text that was phonetically similar to the participants utterances, very few words in the text matched the words actually spoken. In the end, it was easier to create the transcripts manually by listening to the audio than it was to correct the generated transcripts.

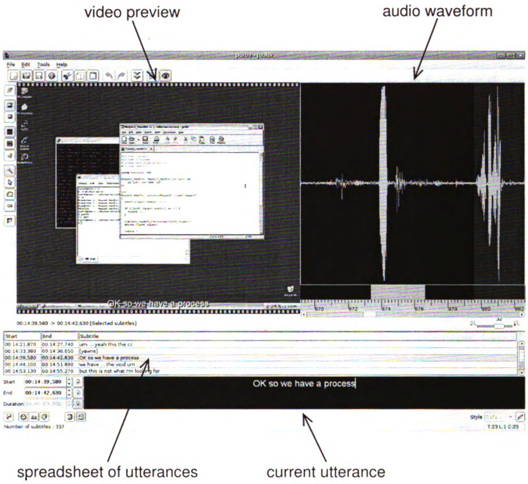
We see two possible causes for the poor quality of transcription. First, the software may have required more training. However, it would not have been reasonable to ask for more of our participants' time. Second, during training participants tended to use a different speaking voice than during their think-aloud sessions. The training software required that participants use a loud, clear voice. However, participants tended to mumble and speak in a lower register while engaged in think-aloud. Unfortunately, trying to make participants speak clearly during think aloud may be distracting to them and degrade their performance.

[6]`http://www.nuance.com/naturallyspeaking/`

In the final analysis, we found the transcription software to be of no help, and performed transcription completely by hand.

Another lesson learned was just how difficult and time-consuming transcription is. We recruited eight computer-science undergraduate students to do the transcribing. Additionally, we transcribed a one-hour and eighteen-minute session ourselves. We kept a log of when and how long we worked on the transcription and found that it took sixteen hours and fifty-three minutes to complete (with an average time of almost thirteen minutes to transcribe each minute of video). We were surprised by how difficult it was to transcribe an utterance correctly without listening to it several times. This need for constant replaying undoubtedly contributed to the long transcription time.

We used the Jubler tool to perform the transcription, and found that several of its featured eased the task. Note that Jubler is intended to be used as a subtitle editor, but its features are also appropriate for transcription. Figure 8.1 depicts the interface of the tool. Jubler's main features are a video-preview window, an audio-waveform window, and a spreadsheet of utterances. By far the must helpful feature was the audio waveform preview window. Clicking on points in the waveform cause the video to jump to that point. Moreover, portions of the waveform can be selected so that the selection can be replayed repeatedly by clicking a button. Another helpful feature was the video-preview window, which conveniently provided contextual information during transcription. The spreadsheet of utterances also incorporated helpful features. The time offset and duration of each utterance is automatically recorded based on the currently selected portion of the waveform. Also, selecting an utterance from the spreadsheet causes the video preview and the waveform to jump to the appropriate time offset. Although manual transcription inherently tedious and labor-intensive, these features effectively reduce some of the effort.



00:14:39,580 -> 00:14:42,630 [Selected subtitles]

Start	End	Subtitle
00:14:21,870	00:14:27,740	um ... yeah this the cc
00:14:33,380	00:14:36,650	(yawns)
00:14:39,580	00:14:42,630	OK so we have a process
00:14:44,100	00:14:51,890	we have ... the void um ...
00:14:53,130	00:14:55,270	but this is not what I'm looking for

Start 00:14:39,580
End 00:14:42,630

OK so we have a process

Number of subtitles : 337

Figure 8.1: Jubler subtitle editor.

8.3 Future Work

Our findings raise interesting questions for future research. One such question is whether failure-trace modeling continues to be a strong predictor of success when the defect is difficult to localize (i.e., to isolate to a segment of code). Two features of the exploratory study aided in localizing the defect. First, participants were able to reproduce the failure, albeit with some difficulty. Second, the bug report included an error message that could be traced directly to a specific line of code. Real-world debugging contexts may not have these features, making the defect considerably more difficult to localize. In future work, we will study how programmers cope with these challenges. Our future study will share the basic design of the exploratory study, except that no error output will be provided and the failure will be effectively unreproducible without manipulating the thread scheduler. In this study, we will look for the localization strategies that lead to success and test whether the ability to localize the defect is a stronger predictor of success than failure-trace modeling.

Another interesting question raised by our work relates to the factors that lead to the breakdown in the systematic management of hypotheses regarding the cause of the failure. We observed that participants in our study failed to analyze many plausible hypotheses. One possible reason for this is that cognitive strain caused them to forget to analyze hypotheses, which they noticed during the course of debugging. Another possible reason is that they had difficulty identifying certain hypotheses in the first place. These potential causes are not mutually exclusive, and we will seek to discover which of them has the greater effect.

We will address this question with a series of controlled experiments. One experiment will compare the use of external representations of hypotheses with the use of strictly internal representations. The experiment will have two treatment groups: an internal group that is asked to use only internal representations and an external group that is asked to externalize hypotheses. We will observe each group as they perform several debugging tasks. To measure how many hypotheses each group analyzes, we will prompt them to think aloud.

In

lo

re

gr

gr

pr

ex

a

e

s

In our analysis, we will check whether the internal group, which is under a heavy cognitive load, analyzes significantly fewer hypotheses than the external group, which has external representations to reduce their load.

Another experiment will evaluate the benefits of possessing a program dependence graph (PDG) [93] (i.e., a graph containing all the data and control dependences of a program) with respect to identifying hypotheses. If formulating hypotheses is a significant problem, a PDG will provide additional information on which to base hypotheses. The experiment will have two treatment groups: a PDG group that is given access to PDGs and a no-PDG group that is not. Each group will perform several debugging tasks. Using the external representations they produce, we will analyze whether the PDG group identifies significantly more hypotheses than the no-PDG group.

CHAPTER 9

CONCLUSIONS

Debugging concurrent software is notoriously difficult. Techniques that programmers depend on to successfully debug sequential software are ineffective when applied to concurrent software. Furthermore, tools have failed to address the problem, and the literature provides little advice about what techniques are effective.

The motivation of the work described in this dissertation is to address the problem by understanding the strategies and practices that successful programmers use when debugging. Such an understanding will both provide advice for programmers and suggest effective tool designs. Toward this goal, we conducted empirical studies of programmers debugging concurrent software applications. Specifically, we ran an exploratory think-aloud study in which we observed fifteen programmers debugging a small multithreaded server application, which we seeded with a defect. Consistent with its exploratory design, this study produced a number of theories regarding the behaviors that distinguish successful from unsuccessful programmers. We conducted a follow-up experiment to test and refine one of these theories.

Three key claims emerged from our studies. The first claim is that successful programmers use the previously-undocumented failure-trace modeling strategy while debugging. The strategy involves modeling interactions among various threads in the system with the objective of discovering a failure trace—that is, an interaction that ends in an error. Finding a failure trace aids in diagnosing the defect because it explains how the program fails. Participants who were successful in our exploratory study were significantly more likely to engage in this strategy than those who were unsuccessful. However, we observed limitations of the strategy: two participants who used it were completely unsuccessful, and many others were unable to model an actual failure trace. Cognitive strain may have been an important contributor to these limitations because participants predominantly modeled internally.

The second claim is that external representations during failure-trace modeling improves the rate of success with the strategy. This claim emerged from our controlled experiment. We compared the ability of programmers who create external representations in the form of UML sequence diagrams to reason about the potential behavior of a concurrent program with that of programmers who use exclusively internal representations. We found that the "external" participants were significantly more successful at correctly reasoning about program behavior than the "internal" ones. Furthermore, our data also suggest that the benefit of external representations increases as the complexity of the program behavior increases.

The third claim is that concurrency makes systematically managing hypotheses regarding the cause of a defect more difficult. This claim is supported by the data of our exploratory study, which suggests that many participants attempted to take a breadth-first approach to diagnosing the defect. However, the approach consistently broke down, leaving many potential causes of the defect unexplored. The complexity of concurrent software causes an explosion in the number of potential causes of a defect. Participants predominantly managed hypotheses internally, and the high volume of hypotheses may have strained their cognitive resources, causing them to forget hypotheses. We also found evidence that concurrency makes hypotheses difficult to discover in the first place. Participants were much less likely to analyze hypotheses that involved reasoning about interactions among multiple threads than they were to analyze hypotheses that only involved reasoning about one thread.

In conclusion, the claims produced by our work represent a good first step toward our long-term goal of understanding the debugging strategies and practices of successful programmers. These claims show great potential for informing the design of new tools as well as educational curricula. Carrying this work forward, we will seek to develop new claims with exploratory studies of different types of applications, defects, and programmers; and we will continue to test and refine our existing claims with controlled experiments and case studies. Moreover, we will develop and evaluate tools that follow directly from our empirical observations. We believe that by grounding debugging solutions in an understanding of how programmers successfully debug concurrent software in practice, we will eventually provide real relief from this vexing problem.

APPENDIX A

EXPLORATORY STUDY MATERIALS

This appendix presents the materials used in our exploratory study. Section A.1 contains the server application, which we seeded with a synchronization defect. Sections A.2 and A.3 contain reproductions of the prestudy and poststudy questionnaires, respectively.

A.1 eBizSim Source Code

Dispatcher.h

```
// -*- C++ -*-

#ifndef SERVER_DISPATCHER_H
#define SERVER_DISPATCHER_H

#include "Pool.h"

class Dispatcher
{
public:
  Dispatcher(Pool* handler_pool);
  virtual ~Dispatcher() {}

  virtual int run();

private:
  Pool* handler_pool_;
};

```

```
#endif /* not SERVER_DISPATCHER_H */
```

Dispatcher.cc

```
// -*- C++ -*-

#include "Dispatcher.h"

#include "Request.h"

using namespace std;

Request* accept_request();

Dispatcher::Dispatcher(Pool* handler_pool)
  : handler_pool_(handler_pool)
{}

int
Dispatcher::run()
{
  while (true) {
    handler_pool_->submit_request(accept_request());
  }
}
```

Request.h

```
// -*- C++ -*-

#ifndef SERVER_REQUEST_H
#define SERVER_REQUEST_H

#include <string>

class Request
{
public:
  virtual ~Request() {}

  // Returns the number of bytes of input or -1 on failure.
  virtual int parse() = 0;

  // Returns a string representation of the request.
  virtual const std::string& to_string() const = 0;
};

#endif /* not SERVER_REQUEST_H */
```

Request_Handler.h

```
// -*- C++ -*-

#ifndef SERVER_REQUEST_HANDLER_H
#define SERVER_REQUEST_HANDLER_H

#include "Request.h"

class Request_Handler
{
public:
  Request_Handler(unsigned id);

  // Returns -1 on failure.
  int process(Request* client_request);

protected:
  void simulate_request_processing(Request* client_request);

private:
  unsigned id_;

  /// Variable for simulating request processing.
  unsigned cur_time_;
};

#endif /* not SERVER_REQUEST_HANDLER_H */
```

Request_Handler.cc

```
// -*- C++ -*-

#include "Request_Handler.h"

```

```
#include <cassert>
#include <iostream>
#include <ace/ACE.h>
#include <ace/OS.h>
#include "Request.h"

using namespace std;

Request_Handler::Request_Handler(unsigned id)
  : id_(id), cur_time_(id)
{}

int
Request_Handler::process(Request* client_request)
{
  assert(client_request != 0);

  if (client_request->parse() == -1) {
    return -1;
  }

  simulate_request_processing(client_request);
  delete client_request;

  return 0;
}

void
Request_Handler::
simulate_request_processing(Request* client_request)
{
  const int SERVICE_TIME[10] = { 1, 1, 0, 2, 1,
                                 0, 1, 1, 0, 2 };
  cur_time_ += id_;
  cur_time_ %= 10;
  ACE_OS::sleep(SERVICE_TIME[cur_time_]);
  cout << "HANDLER " << id_ << ' '
       << "PROCESSED REQUEST: " << client_request->to_string()
       << endl;
}
```

Pool.h

```
// -*- C++ -*-

```

```
#ifndef SERVER_REQUEST_HANDLER_POOL_H
#define SERVER_REQUEST_HANDLER_POOL_H

#include <deque>
#include <vector>
#include <ace/ACE.h>
#include <ace/Thread_Mutex.h>
#include <ace/Condition_Thread_Mutex.h>
#include "Request.h"
#include "Request_Handler.h"

class Pool
{
public:
  Pool(int max_handlers);
  ~Pool() {}

  int dispatch_request();
  void submit_request(Request* request);

private:
  std::vector<Request_Handler*> pool_;
  ACE_Thread_Mutex pool_lock_;
  ACE_Condition_Thread_Mutex nonempty_pool_cond_;
  unsigned pool_waiters_;

  std::deque<Request*> request_queue_;
  ACE_Thread_Mutex queue_lock_;
  ACE_Condition_Thread_Mutex nonempty_queue_cond_;
  unsigned queue_waiters_;

  void add_handler_to_pool(Request_Handler* handler);
  Request_Handler* retrieve_handler_from_pool();
  Request* retrieve_request_from_queue();
};

#endif /* not SERVER_REQUEST_HANDLER_POOL_H */
```

Pool.cc

```
// -*- C++ -*-

#include "Pool.h"

#include <cstdio>
```

```
#include <iostream>
#include <algorithm>

using namespace std;

Pool::Pool(int max_handlers)
  : nonempty_pool_cond_(pool_lock_), pool_waiters_(0),
    nonempty_queue_cond_(queue_lock_), queue_waiters_(0)
{
  // Allocate handlers.
  for (int i = 0; i < max_handlers; ++i) {
    add_handler_to_pool(new Request_Handler(i + 1));
  }
}

int
Pool::dispatch_request()
{
  // First, retrieve a handler from the pool.
  Request_Handler* handler = retrieve_handler_from_pool();

  if (handler == 0) { return -1; }

  // Second, retrieve a request from the queue.
  Request* request = retrieve_request_from_queue();

  if (request == 0) { return -1; }

  // Third, use the handler to process the request.
  if (handler->process(request) == -1) { return -1; }

  // Fourth, return the handler to the pool.
  add_handler_to_pool(handler);

  return 1;
}

void
Pool::submit_request(Request* request)
{
  queue_lock_.acquire();

  request_queue_.push_back(request);

  if (queue_waiters_) {
    nonempty_queue_cond_.signal();
    --queue_waiters_;
  }

  queue_lock_.release();
}

void
Pool::add_handler_to_pool(Request_Handler* handler)
{
  pool_lock_.acquire();

  pool_.push_back(handler);

  if (pool_waiters_) {
    nonempty_pool_cond_.signal();
    --pool_waiters_;
  }

  pool_lock_.release();
}

Request_Handler*
Pool::retrieve_handler_from_pool()
{
  Request_Handler* handler;

  pool_lock_.acquire();

  if (pool_.empty()) {
    ++pool_waiters_;
    nonempty_pool_cond_.wait();
  }

  if (pool_.empty()) {
    pool_lock_.release();
    return 0;
  }

  handler = pool_.back();
  pool_.pop_back();

  pool_lock_.release();

  return handler;
}

Request*
Pool::retrieve_request_from_queue()
```

```
{
  Request* request;

  queue_lock_.acquire();

  if (request_queue_.empty()) {
    ++queue_waiters_;
    nonempty_queue_cond_.wait();
  }

  if (request_queue_.empty()) {
    queue_lock_.release();
    return 0;
  }

  request = request_queue_.front();
  request_queue_.pop_front();

  queue_lock_.release();

  return request;
}
```

server.cc

```
// -*- C++ -*-

#include <cassert>
#include <iostream>
#include <ace/ACE.h>
#include <ace/OS.h>
#include <ace/Thread_Manager.h>
#include "Dispatcher.h"

using namespace std;

// File-local variables.
namespace
{
  const int MAX_THREADS = 20;
  const int MAX_HANDLERS = 20;

  Pool handler_pool(MAX_HANDLERS);
}

```

```
void*
thread_root(void*)
{
  while (true) {
    if (handler_pool.dispatch_request() == -1) {
      cerr << "error: Pool::dispatch_request() failed\n";
      exit(1);
    }
  }

  return 0;
}

int
ACE_MAIN(int argc, char* argv[])
{
  assert(argc == 1);

  Dispatcher dispatcher(&handler_pool);

  // Spawn threads.
  ACE_Thread_Manager::instance()->spawn_n(MAX_THREADS,
                                         thread_root, 0);

  // Start the dispatcher.
  cout << argv[0] << ": begin accepting requests..." << endl;
  dispatcher.run();

  return 0;
}
```

Makefile

```
# -*- Makefile -*-

CPPFLAGS = -D_REENTRANT `pkg-config --cflags ACE` -I../include
CXXFLAGS = -Wall -g
LIBS = ../lib/librequest.a `pkg-config --libs ACE`

server_SOURCES = \
  Dispatcher.cc \
  Request_Handler.cc \
  Pool.cc \
  server.cc

```

```
server_OBJECTS = $(server_SOURCES:.cc=.o)

all: server

server: $(server_OBJECTS)
	g++ $(CXXFLAGS) -o $@ $(server_OBJECTS) $(LIBS)

clean:
	$(RM) server $(server_OBJECTS)
```

A.2 Prestudy Questionnaire

General knowledge of C++ Questions 1 and 2 below refer to Figure A.1, which defines three classes.

1. Suppose the following variable declarations appeared in a context in which the class definitions in Figure A.1 are visible.

```
A  a;            /* Stmt V1 */
A* aPtr;         /* Stmt V2 */
B  b;            /* Stmt V3 */
B* bPtr = new C; /* Stmt V4 */
C  c = new C;    /* Stmt V5 */
```

   According to the rules of the C++ type system, which of the following statements is correct?

   (a) Statements V1, V3, and V5 will compile correctly, but V2 and V4 will generate an error.

   (b) Statements V1, V2, V3, and V5 will compile correctly, but V4 will generate an error.

   (c) Statements V2, V3, V4, and V5 will compile correctly, but V1 will generate an error.

   (d) Every declaration V1 through V5 is type correct and will compile without error.

   (e) None of the above

2. Which of the following sequences of statements will produce the output:

```
In A::f1()
In B::f2()
In C::vf1()
In C::vf2()
```

   (a) `C c; c.f1(); c.f2(); c.vf1(); c.vf2();`

   (b) `B b; C c; b.f1(); b.f2(); c.vf1(); b.vf2();`

   (c) `B* bPtr=new C; bptr->f1(); bPtr->f2(); bPtr->vf1(); bPtr->vf2();`

   (d) All of the above

   (e) None of the above

```
class A
{
public:
  void f1() { cout << "In A::f1()" << endl; }
  virtual void vf1() { cout << "In A::vf1()" << endl; }
  virtual void vf2() = 0;
};

class B : public A
{
public:
  void f2() { cout << "In B::f2()" << endl; }
  virtual void vf2() { cout << "In B::vf2()" << endl; }
};

class C : public B
{
public:
  void f1() { cout << "In C::f1()" << endl; }
  void f2() { cout << "In C::f2()" << endl; }
  void vf1() { cout << "In C::vf1()" << endl; }
  void vf2() { cout << "In C::vf2()" << endl; }
};
```

Figure A.1: Example C++ code for questions 1 to 2.

Familiarity with the ACE toolkit

3. The ACE class `Thread_Manager` provides two means for spawning threads—individually and in groups. The member function `spawn_n` creates a group of threads and takes two formal parameters, an unsigned integer `numThreads` and a pointer (named `rootFunc`) to a `void*` returning function that takes one parameter of type `void*`. Which of the following statements best describes the protocol by which `spawn_n` assigns work to the threads it creates?

   (a) The `spawn_n` method performs some initialization and then invokes the `rootFunc` function, passing `numThreads` as a parameter to this invocation. It is the programmer's responsibility to write `rootFunc` to explicitly spawn an individual thread for each unit of work and to clean up any resources used by these threads once they terminate. The programmer-supplied function determines the number of threads available by casting the `numThreads` parameter from a `void*` to an `unsigned int`. `spawn_n` assumes that `rootFunc` will block until all spawned threads have terminated, unless an error occurs in which case `rootFunc` should return -1. Upon successful termination of all spawned threads `rootFunc` should return 0.

   (b) The `spawn_n` method creates `numThreads` threads, each of which is then immediately dispatched to execute `rootFunc`. The invocation of `spawn_n` then returns, leaving the spawned threads to execute concurrently with the thread that invoked `spawn_n`. Each spawned thread terminates as soon as its execution of `rootFunc` returns or the process exits.

   (c) Same as (b) above, except that the invocation of `spawn_n` waits until all spawned threads have terminated before returning.

   (d) The `spawn_n` method creates a pool of `numThreads` threads, each of which is then dispatched to execute `rootFunc`. As each spawned thread completes its execution of `rootFunc`, it is either terminated or returned to the pool based on the value returned by `rootFunc`. Once returned to the pool, the thread is again dispatched to execute `rootFunc`. This process continues indefinitely. Once all threads have terminated, `spawn_n` returns.

   (e) None of the above.

Prior working knowledge of concurrency concepts Questions 4 through 14 reference ten different scenarios of interaction among concurrent actors that access a shared database object. These actors in this example perform transactions that invoke a series of methods on the database object, and we distinguish two different types of actors—*readers* and *writers*. To illustrate, suppose that the database is storing bank-account information and provides methods for depositing funds, withdrawing funds, and checking the balance of accounts given their account numbers. A typical reader might be interested in computing the total balance of a list of accounts and thus might execute the sequence of operations:

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

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

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

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

Figures A.2, A.3, and A.4 contain the C++ code for the database class, and the reader and writer actors. Notice that the "account management" operations for class `Database` have been elided here for brevity. Class `Database`:

- is implemented according to the monitor-object pattern, using the private variable `lock_` as the monitor lock;
- defines two counting variables, `nReaders_` and `nWriters_`, which record the number of concurrently executing reader and writer transactions respectively; and
- defines two condition variables, `okToRead_` and `okToWrite_`, which are used to synchronize reader and writer threads as they begin and end their transactions.

Please take a moment to familiarize yourself with this code.

```
#include <ace/Thread_Mutex.h>
#include <ace/Condition_Thread_Mutex.h>
```

```
class Database
{
public:
  Database();

  void startRead() const;
  void stopRead() const;

  void startWrite() const;
  void stopWrite() const;

  // ... functions for reading/writing database entries ...

private:
  // The number of readers currently reading
  mutable unsigned nReaders_;

  // The number of writers currently writing
  mutable unsigned nWriters_;

  mutable ACE_Thread_Mutex lock_;
  mutable ACE_Condition_Thread_Mutex okToRead_;
  mutable ACE_Condition_Thread_Mutex okToWrite_;

  // ... data structure for storing entries ...
};
```

Figure A.2: Definition of class Database.

```
Database::Database()
  : nReaders_(0), nWriters_(0),
    okToRead_(lock_), okToWrite_(lock_)
{}

void Database::startRead() const
{
  lock_.acquire();
  while (nWriters_ > 0) okToRead_.wait();
  ++nReaders_;
  lock_.release();
}

void Database::stopRead() const
{
  lock_.acquire();
  --nReaders_;
  if (nReaders_ == 0) okToWrite_.signal();
  lock_.release();
}

void Database::startWrite() const
{
  lock_.acquire();
  while (nWriters_ > 0 || nReaders_ > 0) okToWrite_.wait();
  ++nWriters_;
  lock_.release();
}

void Database::stopWrite() const
{
  lock_.acquire();
  --nWriters_;
  okToWrite_.signal();
  okToRead_.broadcast();
  lock_.release();
}
```

Figure A.3: Implementation of the `Database` member functions.

```
// Global database object
Database db;
```

```
void* reader(void*)
{
  for (;;) {
    db.startRead();
    // ... perform some read operations on db ...
    db.stopRead();
  }
}
```

```
void* writer(void*)
{
  for (;;) {
    db.startWrite();
    // ... perform some read/write operations on db ...
    db.stopWrite();
  }
}
```

```
#include <ace/Thread_Manager.h>
```

```
int main(int, char*[])
{
  ACE_Thread_Manager::instance()->spawn_n(2, writer, 0);
  ACE_Thread_Manager::instance()->spawn_n(2, reader, 0);
  ACE_Thread_Manager::instance()->wait();

  return 0;
}
```

Figure A.4: Implemenation of the reader and writer actors, and the `main` function.

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

4. Shortly thereafter:
   (a) The writer thread obtains the monitor lock.
   (b) The reader thread suspends but does not release the lock.
   (c) The writer thread suspends.
   (d) Both reader and writer threads suspend, and deadlock occurs.
   (e) None of the above

Scenario 2 (an interaction involving one reader and one writer): Assume the reader thread is in the running state and the writer thread is suspended inside the call to `okToWrite_`. The reader thread invokes `stopRead()` and enters the monitor. It sets the `nReaders_` to 0 and issues the call `okToWrite_.signal()`.

5. As a result:
   (a) The writer thread remains suspended.
   (b) The writer thread transitions to ready and acquires the monitor lock.
   (c) The writer thread transitions to ready but does not yet acquire the monitor lock.
   (d) The writer thread transitions to running and acquires the monitor lock.
   (e) The writer thread transitions to running and does not acquire the monitor lock.

6. Upon completing the invocation `okToWrite_.signal()`, the reader thread:
   (a) Must change state to ready and release the monitor lock
   (b) Must change state to ready and retain the monitor lock
   (c) May remain running and must retain the monitor lock
   (d) Must remain running and may release the monitor lock
   (e) Must suspend and release the monitor lock

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

7. Shortly after the writer thread issues the call to `startWrite()`:
   (a) The writer thread remains running until the invocation completes.
   (b) The writer thread suspends on the monitor lock.
   (c) The writer thread obtains the monitor lock but then suspends shortly thereafter.
   (d) The reader thread suspends.
   (e) Deadlock occurs.

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

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

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

9. What will happen as a result of this invocation of `startRead()`?
   (a) r2 enters the monitor but suspends on `okToRead_.wait()`, since `nReaders_` is non-zero at the time.
   (b) r2 enters the monitor but suspends on `okToRead_.wait()`, since `nWriters_` is non-zero at the time.
   (c) r2 suspends on the monitor lock.
   (d) r2 enters the monitor and increases `nReaders_` to two.
   (e) r2 suspends due to the occurrence of a context switch.

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

10. When the reader thread returns from `startRead()`, thus releasing the monitor lock:
   (a) The reader thread must transition to ready; the writer thread must transition to running.
   (b) The reader thread must transition to suspended; the writer thread must transition to running.
   (c) The reader thread may remain running; the writer thread must remain suspended.
   (d) The reader thread may remain running; the writer thread must transition to ready.
   (e) Deadlock occurs.

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

11. Shortly thereafter:
    (a) The reader thread completes the invocation of `startRead()` without suspending.
    (b) The writer suspends because the reader has entered the monitor.
    (c) The reader thread suspends on the monitor lock.
    (d) The reader thread suspends on `okToRead_.wait()`.
    (e) Both (b) and (d) are true; thus deadlock occurs.

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

12. As a result of the signal:
    (a) The reader thread releases the monitor lock; the writer thread transitions to running.
    (b) The reader thread releases the monitor lock; the writer thread transition to ready.
    (c) The reader thread retains the monitor lock; the writer thread remains suspended.
    (d) The reader thread retains the monitor lock; the writer thread transitions to ready.
    (e) Deadlock occurs.

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

13. Shortly thereafter:
    (a) r2 enters the monitor but suspends because `nReaders_` is non-zero when r2 enters the monitor.
    (b) r2 enters the monitor and increases `nReaders_` to two.
    (c) r2 suspends on the monitor lock.
    (d) r2 completes its invocation of `startRead()` without suspending.
    (e) Deadlock occurs.

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

14. As a result of this invocation of `startWrite()`:
    (a) w2 enters the monitor and suspends on `okToWrite_`, retaining the monitor lock.
    (b) w2 enters the monitor and suspends on `okToWrite_`, releasing the monitor lock.
    (c) w2 suspends on the monitor lock.
    (d) w2 enters the monitor and increases `nWriters_` to two.
    (e) Deadlock occurs.

The following questions are not related to any scenario.

15. What feature(s) of the monitor implementation of class `Database` prevent race conditions in updating counting variables (`nReaders_` and `nWriters_`)?
    (a) calls to wait on the condition variables `okToRead_` and `okToWrite_`.
    (b) calls to signal or broadcast on the condition variables `okToRead_` and `okToWrite_`.
    (c) the fact that calls to wait implicitly release the lock before the calling thread suspends.
    (d) the need for any thread to acquire the monitor lock before entering the monitor.
    (e) None of the above.

16. What feature(s) of the monitor implementation prevent a writer from entering the Database while reader(s) are present?
    (a) calls to wait on the condition variables `okToRead_` and `okToWrite_`.
    (b) calls to signal or broadcast on the condition variables `okToRead_` and `okToWrite_`.
    (c) the fact that calls to wait implicitly release the lock before the calling thread suspends.
    (d) the need for any thread to acquire the monitor lock before entering the monitor.
    (e) None of the above.

17. Why does the wait method release the lock and then acquire it again?
    (a) To "wake-up" a reader thread that was previously blocked on the wait.
    (b) To "wake-up" a writer thread that was previously blocked on the wait.
    (c) To promote efficiency.
    (d) To prevent deadlock.
    (e) None of the above.

A.3 Poststudy Questionnaire

Questions related to the specific program under study.

1. Once the eBizSim server is initialized and accepting requests, how many instances of each of the following classes are there at any given time? (Give a number or a "?" if the number varies.)

    1.1. ______ `ACE_SOCK_Stream`

    1.2. ______ `Listener_Socket`

    1.3. ______ `Dispatcher`

    1.4. ______ `Request_Handler`

    1.5. ______ `Request_Handler_Pool`

2. In the eBizSim server, each and every thread plays one of two distinct roles. Think up a name for and briefly describe each role by explaining the responsibilities assumed by any thread that plays it.

3. For each role that you listed in Question 2, please answer:

    - In the normal execution of this system, will there ever be more than one thread playing this role?
    - If more than one, then how many?

4. Select from the following list the server classes whose instances might be concurrently accessed by multiple threads. (Select all that apply.)

    (a) `Listener_Socket`

    (b) `Dispatcher`

    (c) `Request_Handler`

    (d) `Request_Handler_Pool`

    (e) None of the above

5. Consider the Request_Handler_Pool class.

   5.1. Briefly describe the purpose of the two main data structures encapsulated by class Request_Handler_Pool.

   5.2. Might any of these data structures be accessed concurrently by multiple threads? For each such data structure, name the thread role(s) (from Question 2 that are involved.

6. Briefly describe the life of a Request_Handler object in the system (e.g., when is it created, what does it do during its life, and when is it destroyed).

7. Briefly list the major activities performed during an invocation of the operation dispatch_request() in class Request_Handler_Pool in the normal case, i.e., assuming that there are no errors.

8. Of the activities you listed in Question 7, during which of these activities might the actor block? For each such activity, explain the conditions under which the actor will block and the synchronization objects and operations that are involved in implementing this behavior.

9. Consider the scenario in which a thread, call it T, after beginning execution of

   ```
   Request_Handler_Pool::dispatch_request()
   ```

   successfully retrieves a request handler from the pool. For each of the following cases, list all of the activities in Figure A.5 that may then occur at this point in the scenario:

   Case I: the request queue is empty.

   Case II: the request queue has 1 request.

   Case III: the request queue has 10 requests.

10. For each of the following cases, list all of the activities in Figure A.6 that may occur, assuming that T and U are distinct threads.

    Case I: when the activity begins, T holds the lock on request_queue_.

    Case II: when the activity begins, T is waiting on the condition variable nonempty_queue_cond_.

(a) *T* locks `request_queue_` and then waits on `nonempty_queue_cond_`.

(b) Another thread, *U*, locks `request_queue_` and then waits on `nonempty_queue_cond_`.

(c) *T* locks `request_queue_`, retrieves a request, and then releases the lock.

(d) Another thread, *U*, locks `request_queue_`, retrieves a request, and then releases the lock.

(e) Another thread, *U*, locks `request_queue_`, submits a request, and then releases the lock.

(f) Another thread, *U*, locks `request_queue_`, submits a request, calls

```
nonempty_queue_cond_.signal(),
```

and then releases the lock.

Figure A.5: Activities for Question 9.

(a) *T* retrieves a request from `request_queue_`.

(b) *U* retrieves a request from `request_queue_`.

(c) *U* submits a request to `request_queue_`.

(d) *U* retrieves a request handler from `handler_pool_`.

(e) *U* returns a request handler to `handler_pool_`.

(f) *U* waits on the condition variable `nonempty_queue_cond_`.

(g) *U* signals the condition variable `nonempty_queue_cond_`.

Figure A.6: Activities pertaining to Question 10.

11. Which of the following statements is true?

    (a) As an incoming request arrives over the network, a thread is created and then dispatched to handle that request.

    (b) Initially, `MAX_THREADS` threads are created to handle requests. New threads are then spawned if more than `MAX_THREADS` requests need to be processed concurrently.

    (c) Initially, `MAX_THREADS` threads are created to handle requests. If a request arrives and there are no threads available to handle it (because they are handling other requests) the new arrival is dropped.

    (d) All of the threads that will ever be used to handle requests are created before the first request arrives.

    (e) None of the above.

12. Suppose the constants `MAX_THREADS` and `MAX_HANDLERS` are modified so that `MAX_HANDLERS` < `MAX_THREADS`. Which of the following statements would be true in this case?

    (a) `MAX_THREADS` threads will be created and will then compete (synchronize) with one another for access to a more limited number (`MAX_HANDLERS`) of handlers.

    (b) The call to `spawn_n` will block until at least `MAX_THREADS` requests have arrived.

    (c) An error will occur and the system will exit because there are not enough handlers for all of the available threads.

    (d) The system will deadlock.

    (e) None of the above.

13. Suppose the constants `MAX_THREADS` and `MAX_HANDLERS` are modified so that `MAX_HANDLERS` > `MAX_THREADS`. Which of the following statements would be true in this case?

    (a) Only `MAX_THREADS` handlers will actually be allocated.

    (b) `MAX_THREADS` threads will be created and will, eventually, cycle through all of the available handlers provided more than `MAX_HANDLERS` requests arrive before the system halts.

    (c) An error will occur and the system will exit because there are not enough threads to execute all of the available handlers.

    (d) Once `MAX_THREADS` requests arrive, the system will deadlock.

    (e) None of the above.

14. Which of the following general categories of synchronization flaw best describes the design fault in the eBizSim server?

    (a) Two (or more) threads simultaneously write to a shared data structure, i.e., a *data race* results in corrupted data.

    (b) The wait/signal protocol is flawed in that the calls to signal do not match the calls to wait in the sense that a thread may wait on a condition that will never be signaled.

    (c) The wait/signal protocol is flawed in that the waiting threads incorrectly assume a condition is true when they awake, but the underlying wait/signal mechanisms do not guarantee this assumption.

    (d) Two or more threads are *deadlocked*, meaning each holds locks needed by the others in a cycle of dependency.

    (e) None of the above.

15. In your own words, please describe in detail the design fault that leads to the intermittent failure in the eBizSim server. For example, you should be able to give a concrete scenario that demonstrates the fault.

16. What was the significance of the `stress_tester` speed in causing the server to crash (esp. in relation to the state of the request queue)?

17. In your own words, please describe how you fixed (or would fix) the design flaw that leads to the intermittent failure in the eBizSim server.

A.4 Solutions to the Questionnaires

In this section, we provide answer keys for the prestudy and poststudy questionnaires.

A.4.1 Prestudy Questionnaire

1. e
2. c
3. b
4. c
5. c
6. c
7. c
8. c
9. d
10. d
11. d
12. d
13. c
14. b
15. d
16. a
17. d

A.4.2 Poststudy Questionnaire

1.1. 1

1.2. 20

1.3. 1

2. We refer to the two roles as *listener* and *handler*. Listener threads are responsible for accepting client connections and adding open connections, in the form of `Request` objects, to the request queue. Handler threads are responsible for processing requests. They do so by getting a `Request_Handler` object from the pool, getting a `Request` object from the queue, and using the request handler to process the request.

3. There is 1 listener thread and 20 handler threads.

4. c

5.1. The request handler pool stores `Request_Handler` objects that are used to process requests. The request queue stores incoming requests that need to be processed.

5.2. The request handler pool may be concurrently accessed by handler threads. The request queue may be concurrently accessed by listener and handler threads.

6. `Request_Handler` objects are created when the system initializes. They are used to process requests throughout a run of the program. Finally, they are destroyed when the system terminates.

7. An "normal" invocation of `dispatch_request` performs the following activities:

   (1) Retrieves a request handler from the pool.

   (2) Retrieves a request from the queue.

   (3) The request handler is used to process the request, which is subsequently destroyed.

(4) The request handler is returned to the pool.

8. The actor might block during retrieving the request handler from the pool, during retrieving the request from the queue, or during returning the request handler to the pool. In the first case, it blocks if the pool is empty or being accessed by another actor. The mutex lock will cause the thread to block if another thread already holds it, and the thread will block on the `nonempty_pool_cond_` condition variable if the pool is empty. The second case is the same as the first, except it involves the queue and its associated nonempty-condition variable. In the third case, the thread will only block trying to acquire the lock (i.e., when another thread already holds the lock).

9.I. a, b, e, f

9.II. c, d, e, f

9.III. c, d, e, f

10.I. a, d, e

10.II. b, c, d, e, f, g

11. d

12. a

13. e (because handlers are added and retrieved from the back of the pool)

14. c

15. The defect is that the first if-statement in the retrieve-request method should be a while-loop. The failure occurs when a handler thread H_1 is waiting on the nonempty-queue condition variable. The listener thread adds a request to the queue and H_1 is signaled and transitions from the blocking state to the ready state; however, H_1 has not acquired the lock yet. Before H_1 can be scheduled, another handler thread H_2

acquires the lock and empties the queue. Finally, when H_1 is scheduled, it acquires the lock, returns from `wait`, and enters the second if-statement (because the queue is empty), which causes the program to fail. If H_1 had waited in a while-loop, it would have waited again when it found the queue empty.

16. The significance of the speed is that it causes the request queue to rapidly alternate between being empty and nonempty. The failure only occurs when the queue goes from empty to nonempty and one handler thread is signaled and at least one handler thread is in the ready state. If the stress tester is too fast, the queue will never go empty. If the stress tester is too slow, all the threads will be waiting, and there will be no active thread to empty the queue.

17. Change the first if-statement in the retrieve-request method to a while-loop.

APPENDIX B

CONTROLLED EXPERIMENT MATERIALS

This appendix presents the materials used in our controlled experiment. Section B.1 contains a reproduction of the preexperiment questionnaire. Sections B.2 and B.3 contain reproductions of the internal-group and external-group versions of the experiment questionnaire, respectively.

B.1 Preexperiment Questionnaire

Questions 1–8 reference seven different scenarios of interaction among concurrent actors that access a shared database object. The actors in this example perform transactions that invoke a series of methods on the database object, and we distinguish two different types of actors—*readers* and *writers*. To illustrate, suppose the database stores bank-account information and provides methods for depositing funds, withdrawing funds, and checking the balance of accounts given their account numbers. A typical reader might be interested in computing the total balance of a list of accounts and thus might execute the sequence of operations:

```
unsigned sum = 0;

for (unsigned i = 0; i < 10; ++i) {
  sum += db.get_balance(accounts[i]);
}
```

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

```
    db.withdraw(acct_1, 50);
    db.deposit(acct_2, 50);
```

It should be safe for multiple reader transactions to execute concurrently because reader clients do not modify the contents of the database object. However, a writer transaction should never execute concurrently with any reader or any other writer transaction. Class Database supports this *readers-writer* style of synchronization by providing four methods—start_read, stop_read, start_write, and stop_write—which reader and writer threads use to signal the start and finish of one of these transactions.

Figures B.1–B.4 depict the C++ code for class Database as well as functions that implement the reader and writer actors. Notice that the "business logic" has been largely elided—for example, the account-management operations for class Database are absent. Class Database:

- defines two counting variables, n_readers and n_writers, which record the number of concurrently executing reader and writer transactions respectively;
- defines a private variable lock, which is used to enforce mutually exclusive access to n_readers and n_writers; and
- defines two condition variables, ok_to_read and ok_to_write, which are used to synchronize reader and writer threads as they begin and end their transactions.

Please take a moment to familiarize yourself with this code.

Scenario 1: A scenario involving the main thread. Assume that the main thread is in the running state and has just executed the calls to `spawn_n` on lines 83 and 84.

1. Which of the following may happen next? (circle one of the following)

   (a) A context switch occurs such that the main thread transitions to the ready state and one of the reader threads starts running at line 34.

   (b) A context switch occurs such that the main thread transitions to the ready state and one of the writer threads starts running at line 73.

   (c) The main thread remains in the running state and executes the call to `wait` on line 85, which causes it to transition to the ready state.

   (d) A context switch occurs such that the main thread transitions to the blocked state and both reader threads transition to the running state.

   (e) The main thread remains in the running state and all the newly spawned threads transition to the blocked state.

Scenario 2: An interaction involving one reader R and one writer W. Assume W is in the running state within an invocation of `start_write` and R is in the ready state. A context switch occurs just after W increments `n_writers` by one. W transitions to the ready state, and R transitions to the running state. R invokes `start_read` and quickly transitions to the blocked state.

2. Why does R enter the blocked state? (circle one of the following)

   (a) R enters the blocked state via the call to `wait` on line 50 because `n_writers` is non-zero at the time.

   (b) Deadlock occurs. Because W holds the lock, R can't possibly acquire the lock.

   (c) R enters the blocked state via the call to `wait` on line 36 because `n_writers` is non-zero at the time.

   (d) R enters the blocked state via the call to `acquire` on line 35.

   (e) R enters the blocked state due to the occurrence of a context switch.

Scenario 3: An interaction involving one reader thread R and one writer thread W. Assume R is in the running state, and W is in the blocked state while inside the call to `wait` on line 50. R invokes `stop_read` and acquires the lock. It sets the `n_readers` to 0 and issues the call to `signal` on line 44.

3. As a result of the `signal`: (circle one of the following)

   (a) W remains in the blocked state.

   (b) W transitions to the ready state and acquires the lock.

   (c) W transitions to the ready state but does not yet acquire the lock.

   (d) W transitions to the running state and acquires the lock.

   (e) W transitions to the running state and does not acquire the lock.

4. Upon completing the invocation of `signal` on line 44, R: (circle one of the following)

   (a) Must transition to the ready state and release the lock.

   (b) Must transition to the ready state and retain the lock.

   (c) May remain in the running state and must retain the lock.

   (d) Must remain in the running state and may release the lock.

   (e) Must transition to the blocked state and release the lock.

Scenario 4: An interaction involving one reader thread R and one writer thread W. Assume that after R is in the running state and has just returned from an invocation of `start_read`. A context switch occurs such that R transitions to the ready state, and W transitions to the running state. W invokes `start_write`.

5. Shortly after W issues the call to `start_write`: (circle one of the following)

   (a) W remains running until the invocation completes.

   (b) W transitions to the blocked state within the call to `acquire` on the lock.

   (c) W obtains the lock but shortly thereafter releases the lock and transitions to the blocked state.

   (d) R transitions to the blocked state.

   (e) Deadlock occurs.

Scenario 5: An interaction involving two reader threads R_1 and R_2. Assume the R_1 is in the running state and has just completed the invocation of `start_read`, and R_2 is in the ready state. A context switch occurs such that R_1 transitions to the ready state and R_2 transitions to the running state. R_2 issues a call to `start_read`.

6. What will happen as a result of this invocation of `start_read`? (circle one of the following)

   (a) R_2 acquires the lock and increases `n_Readers` to two.

   (b) R_2 acquires the lock but transitions to the blocked state via the call to `wait` on line 36 because `n_readers` is non-zero at the time.

   (c) R_2 acquires the lock but transitions to the blocked state via the call to `wait` on line 36 because `n_writers` is non-zero at the time.

   (d) R_2 transitions to the blocked state via the call to `acquire` on line 35.

   (e) Deadlock occurs.

Scenario 6: An interaction involving two reader threads R_1 and R_2. Assume R_1 is in the running state within an invocation of `start_read`, and R_2 is in the ready state. A context switch occurs just after R_1 increments `n_readers` by one. R_1 transitions to the ready state, and R_2 transitions to the running state. R_2 then invokes `start_read`.

7. Shortly thereafter: (circle one of the following)

   (a) R_2 acquires the lock but transitions to the blocked state via the call to `wait` on line 36 because `n_readers` is non-zero.

   (b) R_2 acquires the lock but transitions to the blocked state via the call to `wait` on line 36 because `n_writers` is non-zero.

   (c) R_2 acquires the lock and increases `n_readers` to two.

   (d) R_2 transitions to the blocked state via the call to `acquire` on line 35.

   (e) Deadlock occurs.

Scenario 7: An interaction involving two writer threads W_1 and W_2. Assume W_1 is in the running state and has just completed its invocation of `start_write`, and W_2 is in the ready state. A context switch occurs. W_1 transitions to the ready state, and W_2 transitions to the running state. W_2 issues a call to `start_write`.

8. As a result of this invocation of `start_write`: (circle one of the following)

   (a) W_2 acquires the lock and transitions to the blocked state via the call to `wait` on line 50, retaining the lock.

   (b) W_2 acquires the lock and transitions to the blocked state via the call to `wait` on line 50, releasing the lock.

   (c) W_2 transitions to the blocked state via the call to `acquire` on line 49.

   (d) W_2 acquires the lock and increases `n_writers` to two.

   (e) Deadlock occurs.

```
#include <ace/Thread_Mutex.h>
#include <ace/Condition_Thread_Mutex.h>
```

```
class Database
{
public:
  Database();

  void start_read() const;
  void stop_read() const;

  void start_write() const;
  void stop_write() const;

  // ... functions for reading/writing database entries ...

private:
  // The number of readers currently reading
  mutable unsigned n_readers;

  // The number of writers currently writing
  mutable unsigned n_writers;

  mutable ACE_Thread_Mutex lock;
  mutable ACE_Condition_Thread_Mutex ok_to_read;
  mutable ACE_Condition_Thread_Mutex ok_to_write;

  // ... data structure for storing entries ...
};
```

Figure B.1: Class definition for a database that allows concurrent readers. The database is implemented using the ACE library. For the sake of simplicity, we elided the "business logic" (e.g., functions for reading and writing database entries).

```
Database::Database()
  : n_readers(0), n_writers(0),
    ok_to_read(lock), ok_to_write(lock)
{}
```

```
void Database::start_read() const
{
  lock.acquire();
  while (n_writers > 0) ok_to_read.wait();
  ++n_readers;
  lock.release();
}
```

```
void Database::stop_read() const
{
  lock.acquire();
  --n_readers;
  if (n_readers == 0) ok_to_write.signal();
  lock.release();
}
```

```
void Database::start_write() const
{
  lock.acquire();
  while (n_writers > 0 || n_readers > 0) ok_to_write.wait();
  ++n_writers;
  lock.release();
}
```

```
void Database::stop_write() const
{
  lock.acquire();
  --n_writers;
  ok_to_write.signal();
  ok_to_read.broadcast();
  lock.release();
}
```

Figure B.2: Member-function definitions for class `Database`.

```
// Global database object
Database db;
```

```
void* reader(void*)
{
  for (;;) {
    db.start_read();
    // ... perform some read operations on db ...
    db.stop_read();
  }
}
```

```
void* writer(void*)
{
  for (;;) {
    db.start_write();
    // ... perform some read/write operations on db ...
    db.stop_write();
  }
}
```

Figure B.3: Primary control loops for the reader and writer threads. For the sake of simplicity, we elided the invocations of operations that read/write the database.

```
#include <ace/Thread_Manager.h>
```

```
int main(int, char*[])
{
  ACE_Thread_Manager::instance()->spawn_n(2, writer, 0);
  ACE_Thread_Manager::instance()->spawn_n(2, reader, 0);
  ACE_Thread_Manager::instance()->wait();

  return 0;
}
```

Figure B.4: The definition of the main function, which spawns two writer threads and two reader threads.

B.2 Experiment Questionnaire: External-Group Version

About the Test

This test involves answering questions about some hypothetical scenarios of behavior of a concurrent program. We organize the test as follows. First, we describe the concurrent program in question (the source code for which is also provided). Then, we present a series of scenarios. For each scenario, you must *first draw a sequence diagram that represents the scenario* and then tell whether:

- The scenario is *consistent* with the code—that is, nothing happens in the scenario that would be impossible with respect to the source code.
- The scenario results in the program entering a clear *error state*—that is, the scenario causes the program to enter a state that would clearly violate the program's specification.

IMPORTANT: Before answering the questions about each scenario, you should draw a sequence diagram that represents the scenario on one of the provided sheets of paper. You may use the diagram to help you answer the questions. Also:

- Don't forget to turn your diagrams in with your test.
- Be sure to label each diagram with appropriate scenario number.

You will be scored based on the quality of your diagrams.

About the Program

The following questions refer to the server program depicted in Figs. B.5–B.6. The server simulates an e-business server that accepts and processes requests from remote clients. The server comprises multiple threads, each of which plays one of two distinct roles—that of a *listener* or that of a *handler*. 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.

The implementation of the server mainly comprises two functions and one class. One function, `listener`, implements the listener-thread behavior (lines 42–49). The other function, `handler`, implements the handler-thread behavior (lines 50–57). The class, `Request_Queue`, represents the shared request queue that the listener thread uses to pass requests to the handler threads (lines 1–39). Since the request queue is accessed by concurrently executing threads, it is implemented using the monitor-object pattern to prevent the threads from interfering with one another. Consistent with the pattern, the request queue has a monitor lock, `lock` (line 9). In addition to mutual-exclusion synchronization provided by the monitor lock, the request queue uses a condition variable, `nonempty`, to provide condition synchronization (line 10). Specifically, handler threads must conditionally wait when attempting to pull a request off an empty queue; they must wait until there is a request to process.

To keep the size of the source code manageable, we elided a number of implementation details that are not salient to this study. In particular, we elided the definitions of the `Request` class and the `accept_request` and `process_request` functions. Instances of class `Request` represent requests that were received over the client connections. The function `accept_request` (line 46) retrieves an incoming client request over the network, packages information about this request into an instance of class `Request`, and returns a pointer to this instance. A thread that invokes this function when there are no pending client requests on the network will block indefinitely until a request arrives. The function `process_request` (line 55) simulates the servicing of a client request. The function takes a (non-null) pointer to a `Request` as an argument. Once serviced, the function deletes the `Request` object. For the remainder of the test, you should assume that the elided class and functions were implemented correctly and that they behave as described.

Regarding the deque: The instance of `deque` (declared on line 8) represents a *double-ended queue*; however, for our purposes, you may think of this as a regular queue. The class `deque` has the following operations:

`push_back:` Adds an item to the back of the `deque`.

`pop_front:` Removes the item at the front of the `deque` (returns `void`).

`front:` Returns the item at the front of the `deque` (does not modify the `deque`).

`empty:` Returns true if the `deque` is empty; otherwise, false.

About the Scenarios

The scenarios represent sequences of internal program behavior that the e-business server may exhibit during a hypothetical run. The description of a scenario provides just enough information for you to reason about the represented behavior. Specifically, the description includes

- the state of the system when the scenario begins,
- all the calls to and returns from operations of `Request_Queue` during the scenario, and
- some additional information needed to make the scenario unambiguous.

You must infer the details of each thread's behavior during the scenario based on the descriptions given.

Scenario 1

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.

Questions

IMPORTANT: DRAW A SEQUENCE DIAGRAM OF THE ABOVE SCENARIO BEFORE ANSWERING THESE QUESTIONS.

---

1. Is the scenario consistent with the code? If so, does the scenario result in the program entering an error state? (Circle 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.

Scenario 2

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 is preempted at line 29.

(2) H_2 calls `retrieve` and blocks inside the operation.

(3) L calls `submit` (with argument r) and blocks inside the operation.

(4) H_1 transitions to the running state and subsequently blocks (never having returned from `retrieve`). In the process, H_2 transitions to the ready state.

Questions

IMPORTANT: DRAW A SEQUENCE DIAGRAM OF THE ABOVE SCENARIO BEFORE ANSWERING THESE QUESTIONS.

2. Is the scenario consistent with the code? If so, does the scenario result in the program entering an error state? (Circle 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.

Scenario 3

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

- `queue` is empty,
- `waiters` is one,
- L and H_2 are at the beginning of their respective control loops, and
- H_1 is blocking inside the call to `wait` on line 32.

Consider the scenario where:

(1) L calls and returns from `submit` (having passed in argument r), and is preempted at the top of its control loop. In the process, H_1 transitions to the ready state.

(2) H_2 calls and returns from `retrieve`, and is preempted at the top of its control loop.

(3) H_1 returns from `retrieve`, and is preempted at the top of its control loop.

Questions

IMPORTANT: DRAW A SEQUENCE DIAGRAM OF THE ABOVE SCENARIO BEFORE ANSWERING THESE QUESTIONS.

3. Is the scenario consistent with the code? If so, does the scenario result in the program entering an error state? (Circle 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.

Scenario 4

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

- `queue` is empty,
- `waiters` is two,
- L is at the beginning of its control loop, and
- H_1 and H_2 are both blocking inside the call to `wait` on line 32.

Consider the scenario where:

(1) L calls and returns from `submit` (having passed in argument r), and is preempted at the top of its control loop.

(2) H_1 returns from `retrieve` and is preempted at the top of its control loop.

(3) H_2 transitions to the running state and subsequently blocks (never having returned from `retrieve`).

Questions

IMPORTANT: DRAW A SEQUENCE DIAGRAM OF THE ABOVE SCENARIO <u>BEFORE</u> ANSWERING THESE QUESTIONS.

4. Is the scenario consistent with the code? If so, does the scenario result in the program entering an error state? (Circle 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.

```
class Request_Queue
{
public:
  Request_Queue() : nonempty(lock), waiters(0) {}
  void submit(Request* request);
  Request* retrieve();
private:
  deque<Request*> queue;
  ACE_Thread_Mutex lock;
  ACE_Condition_Thread_Mutex nonempty;
  unsigned waiters;
};
```

```
void Request_Queue::submit(Request* request)
{
  lock.acquire();
  queue.push_back(request);

  if (waiters > 0) {
    nonempty.signal();
    --waiters;
  }

  lock.release();
}
```

```
Request* Request_Queue::retrieve()
{
  Request* request;
  lock.acquire();

  if (queue.empty()) {
    ++waiters;
    nonempty.wait();
  }

  request = queue.front();
  queue.pop_front();
  lock.release();
  return request;
}
```

Figure B.5: The request-queue interface and implementation. For the sake of simplicity, we elide the definition of the class, `Request`.

```
// Global request queue
Request_Queue rqueue;
```

```
void listener()
{
  // Listener-thread control loop
  for (;;) {
    Request* request = accept_request();
    rqueue.submit(request);
  }
}
```

```
void* handler(void*)
{
  // Handler-thread control loop
  for (;;) {
    Request* request = rqueue.retrieve();
    process_request(request);
  }
}
```

```
int main(int, char*[])
{
  ACE_Thread_Manager::instance()->spawn_n(2, handler, 0);
  listener();
  return 0;
}
```

Figure B.6: The listener and handler implementations, and the definition of the function, `main`. For the sake of simplicity, we elide the definition the class, `Request`, and the definitions of the functions, `accept_request` and `process_request`.

B.3 Experiment Questionnaire: Internal-Group Version

The only differences between the internal-group version of the questionnaire and the external-group version were in the instructions and the wording of the questions. In this section, we reproduce the internal-group version of the instructions and questions.

About the Test

This test involves answering questions about some hypothetical scenarios of behavior of a concurrent program. We organize the test as follows. First, we describe the concurrent program in question (the source code for which is also provided). Then, we present a series of scenarios. For each scenario, you must tell whether:

- The scenario is *consistent* with the code—that is, nothing happens in the scenario that would be impossible with respect to the source code.
- The scenario results in the program entering a clear *error state*—that is, the scenario causes the program to enter a state that would clearly violate the program's specification.

IMPORTANT: Answer the questions "in your head" without making notes or drawing pictures.

Questions

Is the scenario consistent with the code? If so, does the scenario result in the program entering an error state? (Circle 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.

B.4 Solutions to the Questionnaires

In this section, we provide answer keys for the preexperiment and experiment questionnaires.

B.4.1 Preexperiment Questionnaire Solutions

1. b
2. d
3. c
4. c
5. c
6. a
7. d
8. b

B.4.2 Experiment Questionnaire Solutions

1. a
2. a
3. b
4. c

Figures B.7–B.10 depict sequence diagrams that represent the scenarios from the questionnaire.

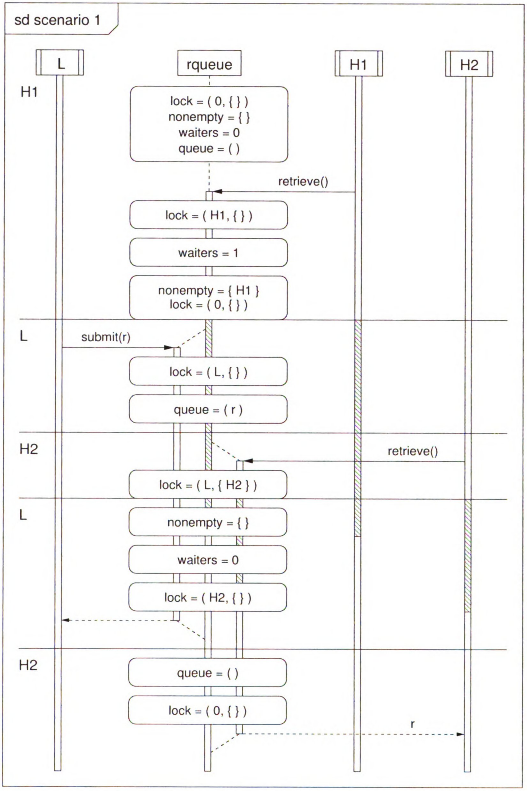


Figure B.7: Sequence diagram for Scenario 1.

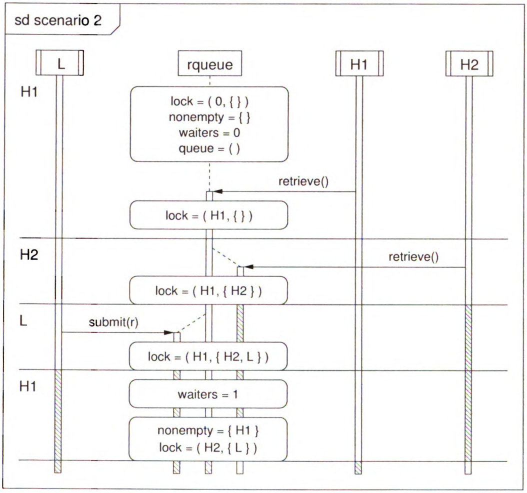


Figure B.8: Sequence diagram for Scenario 2.

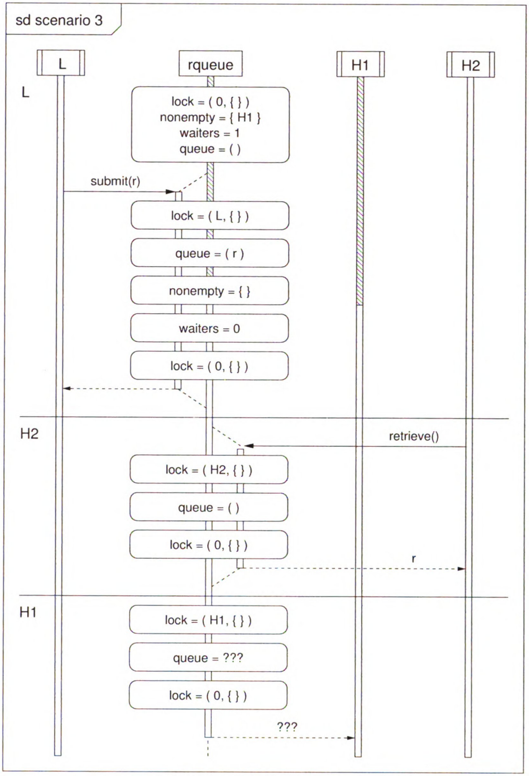


Figure B.9: Sequence diagram for Scenario 3.

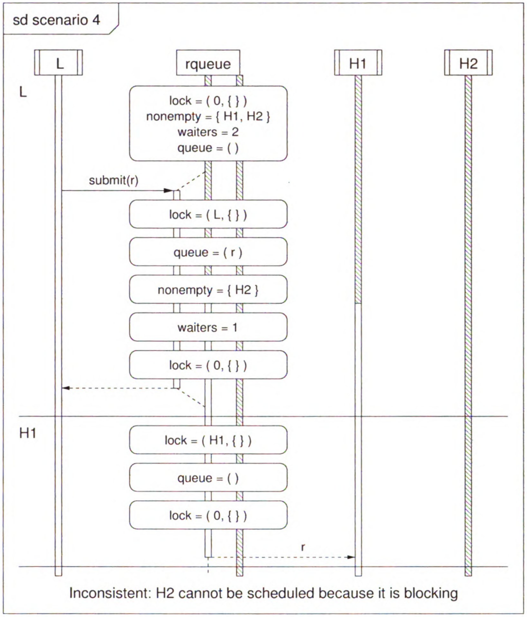


Figure B.10: Sequence diagram for Scenario 4.

B.5 Diagram-Evaluation Rubric

During our analysis, we evaluated each sequence diagram that a participant produced and assigned it a score in the range [0,10]. We checked for the presence and correctness of ten features. For each of the feature, if the feature is completely absent, the score for that feature is 0 by default. Detailed scoring criteria for each feature follow. Deductions for a single feature that amount to more than one point are reduced to -1 (i.e., no more than one point may be deducted per feature).

Feature: Objects. Scoring criteria:

- Missing/conflated objects: -1 deduction
- No active-object bars: no deduction
- Oddly named request queue: no deduction

Feature: Initial State of rqueue. Scoring criteria:

- Unusual representation of states: no deduction
- Two or more missing field(s): -1 deduction

Feature: Calls/returns. Scoring criteria:

- Erroneous ordering: -1 deduction
- Missing calls/returns: -1 deduction

  Exception: if return is implicitly specified by activation bars: no deduction
- Two or more missing labels: -1 deduction
- Calls/returns incorrectly given dashed/solid lines: no deduction
- Missing arrowheads: no deduction

- Missing call arguments or return values: no deduction

Feature: Activations. Scoring criteria:

- Missing/incorrect call activations: -1 deduction
- Passive objects always activated: no deduction
- Active objects not continuously activated: no deduction
- Activations indicated by bar-like notation other than bar: no deduction

Feature: Running thread. Scoring criteria:

- Missing labels as long as it's clear who executes: no deductions

Feature: Lock state changes. Scoring criteria:

- Missing or incorrect changes in lock state: -1 deduction

  Exception: signal places signaled thread in lock's wait-set: no deduction

  Exception: lock release does not move waiter to holder, instead when the next waiter executes it becomes holder: no deduction

Feature: Condition state changes. Scoring criteria:

- Missing or incorrect changes of condition state: -1 deduction
- Split waiting state-change bubble: no deduction

Feature: Queue state changes. Scoring criteria:

- Missing or incorrect changes of queue state: -1 deduction

Feature: Waiters state changes. Scoring criteria:

- Missing or incorrect changes of waiters state: -1 deduction

Feature: Blocking. Scoring criteria:

- Incorrect blocking: -1 deduction

  Exception: Innocuous gaps between wait/lock state changes and the beginning/end of blocking: no deduction

- Blocking in the middle of a split waiting state-change bubble: no deduction

- Some activations not showing blocking while others show it: no deduction

- Use of alternate notation to show blocking: no deduction

Additional scoring criteria.

- If a state is given that does not reflect a change of state, then no deduction is taken

- If multiple state changes are captured in a single bubble, then no deduction is taken

- If there is no graphical bubble around state information, then no deduction is taken

- Some variation in state text is allowed as long as it is clear

- Some variations in notation are allowed as long as they capture all the information

- In the case of a conflated object, treat it as two objects in evaluating the other properties modeling the state of local variables does not result in a deduction

- State variables that are combined/conflated are OK as long as not information is lost

APPENDIX C

FAULT TREE FOR THE EBIZSIM FAILURE

To analyze how the participants in our exploratory study managed hypotheses regarding the cause of the defect, we developed a fault tree [29, 70] for the failure in our study. This appendix contains a fully-elaborated representation of that fault tree. The fault tree has three main subtrees: the L subtree (Figure C.1), the M subtree (Figure C.2), and the R subtree (Figure C.3). Section 4.2 provides a detailed explanation of the notation used in the figures.

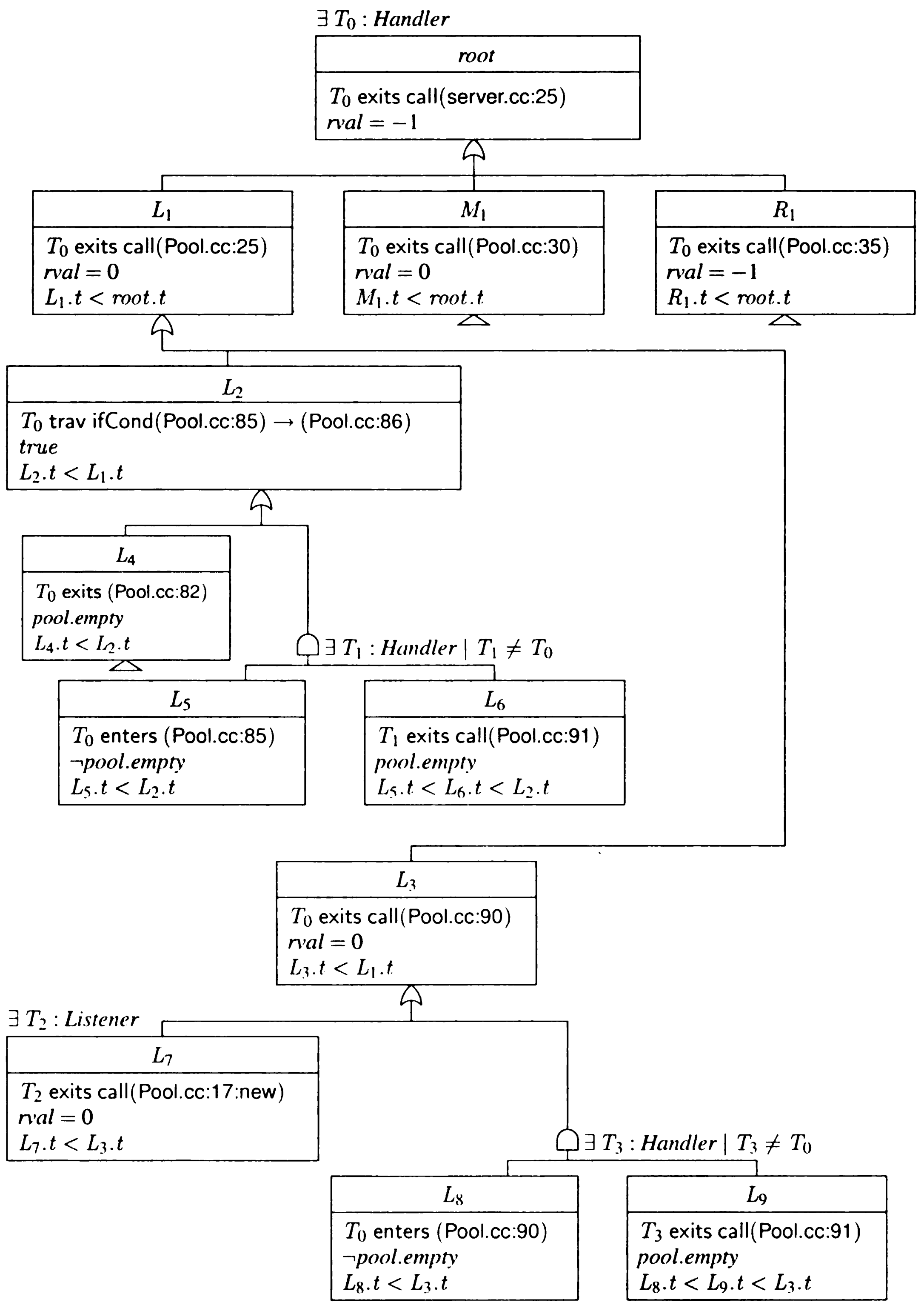


Figure C.1: L subtree of the fault tree for the eBizSim failure.

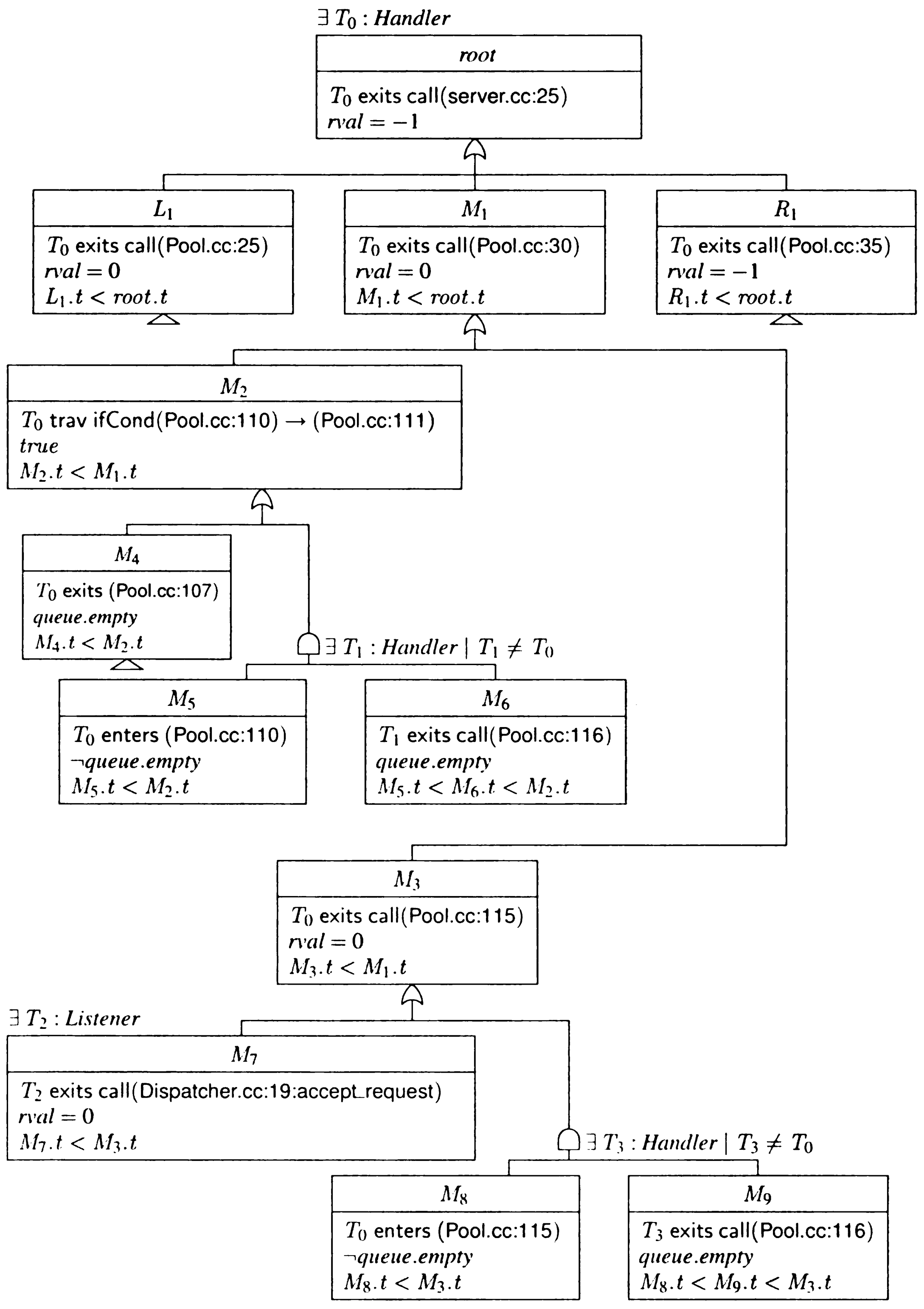
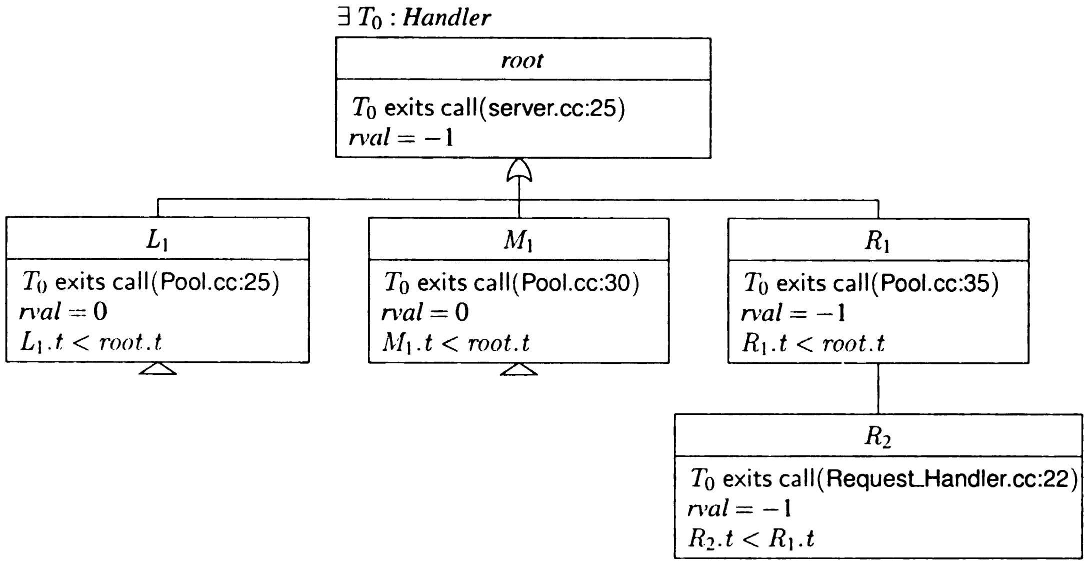

$\exists T_0 : Handler$
$root$
T_0 exits call(server.cc:25)
$rval = -1$
L_1
T_0 exits call(Pool.cc:25)
$rval = 0$
$L_1.t < root.t$
M_1
T_0 exits call(Pool.cc:30)
$rval = 0$
$M_1.t < root.t$
R_1
T_0 exits call(Pool.cc:35)
$rval = -1$
$R_1.t < root.t$
M_2
T_0 trav ifCond(Pool.cc:110) $\rightarrow$ (Pool.cc:111)
$true$
$M_2.t < M_1.t$
M_4
T_0 exits (Pool.cc:107)
$queue.empty$
$M_4.t < M_2.t$
$\exists T_1 : Handler \mid T_1 \neq T_0$
M_5
T_0 enters (Pool.cc:110)
$\neg queue.empty$
$M_5.t < M_2.t$
M_6
T_1 exits call(Pool.cc:116)
$queue.empty$
$M_5.t < M_6.t < M_2.t$
M_3
T_0 exits call(Pool.cc:115)
$rval = 0$
$M_3.t < M_1.t$
$\exists T_2 : Listener$
M_7
T_2 exits call(Dispatcher.cc:19:accept_request)
$rval = 0$
$M_7.t < M_3.t$
$\exists T_3 : Handler \mid T_3 \neq T_0$
M_8
T_0 enters (Pool.cc:115)
$\neg queue.empty$
$M_8.t < M_3.t$
M_9
T_3 exits call(Pool.cc:116)
$queue.empty$
$M_8.t < M_9.t < M_3.t$

Figure C.2: M subtree of the fault tree for the eBizSim failure.



$\exists\, T_0$: *Handler*
root
T_0 exits call(server.cc:25)
$rval = -1$
L_1
T_0 exits call(Pool.cc:25)
$rval = 0$
$L_1.t < root.t$
M_1
T_0 exits call(Pool.cc:30)
$rval = 0$
$M_1.t < root.t$
R_1
T_0 exits call(Pool.cc:35)
$rval = -1$
$R_1.t < root.t$
R_2
T_0 exits call(Request_Handler.cc:22)
$rval = -1$
$R_2.t < R_1.t$

Figure C.3: *R* subtree of the fault tree for the eBizSim failure.

APPENDIX D

INTERACTION COMPLEXITY ANALYSIS

In this section, we provide the artifacts that resulted from our complexity analysis of the scenarios from the questionnaire. First, we list the FSP code that models the questionnaire program as an LTS. Second, we list the traces of the model that are associated with each scenario from the questionnaire.

D.1 LTS Model of Questionnaire Program

We modeled the program from the experiment questionnaire as an LTS. The FSP that represents the LTS follows.

```
const MAX_THREADS = 3
set ALL_THREADS = { h1, h2, l }
set HANDLER_THREADS = { h1, h2 }
set LISTENER_THREADS = { l }
set THE_LOCK_THREADS = { h1, h2, l }
set NONEMPTY_THREADS = { h1, h2, l }
set QUEUE_THREADS = { h1, h2, l }
set WAITERS_THREADS = { h1, h2, l }

LOCK = FREE_LOCK ,
FREE_LOCK = ( acquire -> HELD_LOCK[0] ) ,
HELD_LOCK[i:0..MAX_THREADS] =
  ( when (i == 0) release -> FREE_LOCK
  | when (i == 0) try_acquire_block -> HELD_LOCK[1]
  | when (i > 0 && i <= MAX_THREADS)
      release -> unblock_acquire -> HELD_LOCK[i-1]
  | when (i > 0 && i < MAX_THREADS)
      try_acquire_block -> HELD_LOCK[i+1]
  ) .

COND = COND_WAITERS[0] ,
COND_WAITERS[i:0..MAX_THREADS] =
  ( when (i == 0) {signal, broadcast} -> COND_WAITERS[0]
  | when (i < MAX_THREADS) wait -> COND_WAITERS[i+1]
  | when (i > 0 && i <= MAX_THREADS)
      signal -> end_wait -> COND_WAITERS[i-1]
  | when (i > 0 && i <= MAX_THREADS)
      broadcast -> COND_WAKE_ALL[i]
  ) ,
COND_WAKE_ALL[j:1..MAX_THREADS] =
  ( when (j == 1) end_wait -> COND_WAITERS[0]
  | when (j > 1 && j <= MAX_THREADS)
      end_wait -> COND_WAKE_ALL[j-1]
  ) .

const MAXCTR = 3

COUNTER(MAX=MAXCTR) = CTR[0],
CTR[i:0..MAX] = (read[i] -> CTR[i]
                | when (i<MAX) inc -> CTR[i+1]
                | when (i>0) dec -> CTR[i-1]) .

set LOCK_OPS = { acquire, try_acquire_block,
                 unblock_acquire, release }
set LOCKS = { the_lock }
set CONDITION_VAR_OPS = { wait, signal, broadcast, end_wait }
set CONDITION_VARS = { nonempty }
set COUNTER_OPS = { read[0..MAXCTR], inc, dec }
set COUNTERS = { queue, waiters }
set DATA_MEMBER_CALLS =
  { LOCKS.LOCK_OPS,
    CONDITION_VARS.CONDITION_VAR_OPS,
    COUNTERS.COUNTER_OPS
  }
const MAX_QUEUE = 2
range QUEUE_RANGE = 0..MAX_QUEUE
const MAX_WAITERS = 2
range WAITERS_RANGE = 0..MAX_WAITERS

||THE_LOCK = the_lock:LOCK .
||NONEMPTY = nonempty:COND .
||QUEUE = queue:COUNTER(MAX_QUEUE) .
||WAITERS = waiters:COUNTER(MAX_WAITERS) .
```

```

NONEMPTY_WAIT = ( nonempty.wait -> the_lock.release ->
                  nonempty.end_wait -> END )
                + {nonempty.signal, nonempty.broadcast} .
NONEMPTY_SIGNAL = ( nonempty.signal -> END )
                  + {nonempty.wait, nonempty.end_wait,
                     nonempty.broadcast} .
NONEMPTY_BROADCAST = ( nonempty.broadcast -> END )
                     + {nonempty.wait, nonempty.end_wait,
                        nonempty.signal} .

QUEUE_DEC = (queue.dec -> END)
            + { queue.inc, queue.read[QUEUE_RANGE] } .
QUEUE_INC = (queue.inc -> END)
            + { queue.dec, queue.read[QUEUE_RANGE] } .

THE_LOCK_ACQUIRE = ( the_lock.acquire -> END
                   | the_lock.try_acquire_block ->
                     the_lock.unblock_acquire -> END ) .
THE_LOCK_RELEASE = ( the_lock.release -> END ) .

WAITERS_DEC = (waiters.dec -> END)
              + { waiters.inc, waiters.read[WAITERS_RANGE] }.
WAITERS_INC = (waiters.inc -> END)
              + { waiters.dec, waiters.read[WAITERS_RANGE] }.

IF_1_1 =
  ( waiters.read[n1:WAITERS_RANGE] ->
    if (n1>0)
    then NONEMPTY_SIGNAL;
         WAITERS_DEC;
         END
    else END
  ) .

IF_2_1 =
  ( queue.read[n2:QUEUE_RANGE] ->
    if (n2==0)
    then WAITERS_INC;
         NONEMPTY_WAIT;
         THE_LOCK_ACQUIRE;
         END
    else END
  ) .

RETRIEVE_CALL = ( call_retrieve -> END ) .
RETRIEVE_RETURN = ( return_from_retrieve -> END ) .
```

```
RETRIEVE_METHOD = RETRIEVE_CALL
                ; THE_LOCK_ACQUIRE
                ; IF_2_1
                ; QUEUE_DEC
                ; THE_LOCK_RELEASE
                ; RETRIEVE_RETURN
                ; END
                .

SUBMIT_CALL = ( call_submit -> END ) .
SUBMIT_RETURN = ( return_from_submit -> END ) .
SUBMIT_METHOD = SUBMIT_CALL
              ; THE_LOCK_ACQUIRE
              ; QUEUE_INC
              ; IF_1_1
              ; THE_LOCK_RELEASE
              ; SUBMIT_RETURN
              ; END
              .

||REQUESTQUEUE_AS_A_SHARED_RESOURCE =
  (  THE_LOCK_THREADS::THE_LOCK
  || NONEMPTY_THREADS::NONEMPTY
  || QUEUE_THREADS::QUEUE
  || WAITERS_THREADS::WAITERS
  ) .

HANDLER = RETRIEVE_METHOD
        ; END
        .

LISTENER = SUBMIT_METHOD
         ; SUBMIT_METHOD
         ; END
         .

||PROG =
  (  HANDLER_THREADS:HANDLER
  || LISTENER_THREADS:LISTENER
  || REQUESTQUEUE_AS_A_SHARED_RESOURCE
  ) .
```

D.2 Scenario Traces

To analyze the complexity of the interactions represented by the scenarios, we produced a trace of the LTS for each scenario. Tables D.1–D.4 depict these traces. Each table lists the sequence of actions, the thread that executed each action, and the actions that represent block/unblock actions. Note that traces #3 and #4 do not begin in the initial state of the program.

Table D.1: Scenario #1 trace.

Number	Action	Actor	Block or Unblock
1	h1.call_retrieve	H_1	
2	h1.the_lock.acquire	H_1	
3	h1.queue.read.0	H_1	
4	h1.waiters.inc	H_1	
5	h1.nonempty.wait	H_1	Block
6	h1.the_lock.release	H_1	
7	l.call_submit	L	
8	l.the_lock.acquire	L	
9	l.queue.inc	L	
10	h2.call_retrieve	H_2	
11	h2.the_lock.try_acquire_block	H_2	Block
12	l.waiters.read.1	L	
13	l.nonempty.signal	L	
14	h1.nonempty.end_wait	L	Unblock
15	l.waiters.dec	L	
16	l.the_lock.release	L	
17	h2.the_lock.unblock_acquire	L	Unblock
18	l.return_from_submit	L	
19	h2.queue.read.1	H_2	
20	h2.queue.dec	H_2	
21	h2.the_lock.release	H_2	
22	h2.return_from_retrieve	H_2	

Table D.2: Scenario #2 trace.

Number	Action	Actor	Block or Unblock
1	h1.call_retrieve	H_1	
2	h1.the_lock.acquire	H_1	
3	h2.call_retrieve	H_2	
4	h2.the_lock.try_acquire_block	H_2	Block
5	l.call_submit	L	
6	l.the_lock.try_acquire_block	L	Block
7	h1.queue.read.0	H_1	
8	h1.waiters.inc	H_1	
9	h1.nonempty.wait	H_1	Block
10	h1.the_lock.release	H_1	
11	h2.the_lock.unblock_acquire	H_1	Unblock

Table D.3: Scenario #3 trace.

Number	Action	Actor	Block or Unblock
1	l.call_submit	L	
2	l.the_lock.acquire	L	
3	l.queue.inc	L	
4	l.waiters.read.1	L	
5	l.nonempty.signal	L	
6	h1.nonempty.end_wait	L	Unblock
7	l.waiters.dec	L	
8	l.the_lock.release	L	
9	l.return_from_submit	L	
10	h2.call_retrieve	H_2	
11	h2.queue.read.1	H_2	
12	h2.queue.dec	H_2	
13	h2.the_lock.release	H_2	
14	h2.return_from_retrieve	H_2	
15	h1.the_lock.acquire	H_1	

Table D.4: Scenario #4 trace.

Number	Action	Actor	Block or Unblock
1	l.call_submit	L	
2	l.the_lock.acquire	L	
3	l.queue.inc	L	
4	l.waiters.read.2	L	
5	l.nonempty.signal	L	
6	h1.nonempty.end_wait	L	Unblock
7	l.waiters.dec	L	
8	l.the_lock.release	L	
9	l.return_from_submit	L	
10	h1.the_lock.acquire	H_1	
11	h1.queue.dec	H_1	
12	h1.the_lock.release	H_1	
13	h1.return_from_retrieve	H_1	

BIBLIOGRAPHY

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison, 1988.

[2] A. Aiken and D. Gay. Barrier inference. In *Proc. 25th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 1998)*, pages 342–354, 1998.

[3] American Psychological Association. *Ethical Principles of Psychologists and Code of Conduct*. APA, 2002.

[4] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin, 1991.

[5] K. Araki, Z. Furukawa, and J. Cheng. A general framework for debugging. *IEEE Softw.*, 8(3):14–20, 1991.

[6] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proc. 1991 ACM/ONR Workshop Parallel and Distributed Debugging (PADD 1991)*, pages 194–206, 1991.

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

[8] V. R. Basili. The role of experimentation in software engineering: Past, current, and future. In *Proc. 18th Int. Conf. Software Eng. (ICSE 1996)*, pages 442–449, 1996.

[9] V. R. Basili, D. Cruzes, J. C. Carver, L. M. Hochstein, J. K. Hollingsworth, M. V. Zelkowitz, and F. Shull. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Softw.*, 25(4):29–36, 2008.

[10] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice, 1990.

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

[12] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proc. 15th Int. Conf. Software Eng. (ICSE 1993)*, pages 482–498, 1993.

[13] B. S. Bloom. *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Addison, 1956.

[14] L. C. Briand, C. Bunse, and J. W. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Trans. Softw. Eng.*, 27(6):513–530, 2001.

[15] L. C. Briand, Y. Labiche, and J. Leduc. Towards the reverse engineering of UML sequence diagrams for distributed, multithreaded Java software. Technical Report SCE-04-04, Carleton Univ., 2004.

[16] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Trans. Softw. Eng.*, 32(9):642–663, 2006.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annu. Symp. Logic in Comput. Sci. (LICS 1990)*, pages 428–439, 1990.

[18] D. R. Butenhof. *Programming with POSIX Threads.* Addison, 1997.

[19] R. H. Carver and K. C. Tai. Reproducible testing of concurrent programs based on shared variables. In *Proc. 6th Int. Conf. Distributed Computing Systems (ICDCS 1986)*, pages 428–433, 1986.

[20] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8(2):66–74, 1991.

[21] Z. Chen, B. Xu, and J. Zhao. An overview of methods for dependence analysis of concurrent programs. *SIGPLAN Not.*, 37(8):45–52, 2002.

[22] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proc. 10th Annu. ACM Symp. Parallel Algorithms and Architectures (SPAA 1998)*, pages 298–309, 1998.

[23] J. Cheng. Slicing concurrent programs: A graph-theoretical approach. In *Proc. 1st Int. Workshop Automated and Algorithmic Debugging (AADEBUG 1993)*, pages 223–240, 1993.

[24] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. ACM SIGPLAN 2002 Conf. Programming Language Design and Implementation (PLDI 2002)*, pages 258–269, 2002.

[25] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proc. SIGMETRICS Symp. Parallel and Distributed Tools (SPDT 1998)*, pages 48–59, 1998.

[26] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, 1981.

[27] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[28] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT P., 1999.

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

[30] M. P. Consens, M. Z. Hasan, and A. O. Mendelzon. Using Hy+ for network management and distributed debugging. In *Proc. 1993 Conf. Centre for Advanced Studies on Collaborative Research (CASCON 1993)*, pages 450–471, 1993.

[31] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd Int. Conf. Software Eng. (ICSE 2000)*, pages 439–448, 2000.

[32] J. Corbin and A. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 3rd edition, 2008.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT P., 2nd edition, 2001.

[34] R. Curtis and L. Wittie. BugNet: A debugging system for parallel programming environments. In *Proc. 3rd Int. Conf. Distributed Computing Systems (ICDCS 1982)*, pages 394–399, 1982.

[35] E. Duesterwald, R. Gupta, and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Proc. 5th Int. Workshop Languages and Compilers for Parallel Computing (LCPC 1992)*, pages 497–511, 1992.

[36] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Eng. Methodol.*, 13(4):359–430, 2004.

[37] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In F. Shull, J. Singer, and D. I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, chapter 11, pages 285–311. Springer, 2008.

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

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

[40] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS, 2nd edition, 1998.

[41] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN 2000 Conf. Programming Language Design and Implementation (PLDI 2000)*, pages 219–232, 2000.

[42] S. D. Fleming, E. Kraemer, R. E. K. Stirewalt, L. K. Dillon, and S. Xie. Refining existing theories of program comprehension during maintenance for concurrent software. In *Proc. 16th IEEE Int. Conf. Program Comprehension (ICPC 2008)*, pages 23–32, 2008.

[43] 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. 30th Int. Conf. Software Eng. (ICSE 2008)*, pages 759–768, 2008.

[44] J. Gait. A probe effect in concurrent programs. *Softw. Pract. Exper.*, 16(3):225–233, 1986.

[45] A. Georges, M. Christiaens, M. Ronsse, and K. D. Bosschere. JaRec: A portable record/replay environment for multi-threaded Java applications. *Softw. Pract. Exper.*, 34(6):523–547, 2004.

[46] J. D. Gould. Some psychological evidence on how people debug computer programs. *Int. J. Man-Mach. Stud.*, 7(2):151–182, 1975.

[47] J. D. Gould and P. Drongowski. An exploratory study of computer program debugging. *Hum. Factors*, 16(3):258–277, 1974.

[48] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proc. 6th Int. Symp. Static Anal. (SAS 1999)*, pages 1–18, 1999.

[49] M. Hibberd, M. Lawley, and K. Raymond. Forensic debugging of model transformations. In *Proc. ACM/IEEE 10th Int. Conf. Model Driven Eng. Languages and Syst. (MoDELS 2007)*, pages 589–604, 2007.

[50] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[51] L. Hochstein and V. R. Basili. An empirical study to compare two parallel programming models. In *Proc. 18th Annu. ACM Symp. Parallelism in Algorithms and Architectures (SPAA 2006)*, pages 114–114, 2006.

[52] L. Hochstein and V. R. Basili. The ASC-Alliance projects: A case study of large-scale parallel scientific code development. *Computer*, 41(3):50–58, 2008.

[53] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *J. Syst. Softw.*, 81(11):1920–1930, 2008.

[54] L. Hochstein, V. R. Basili, M. V. Zelkowitz, J. K. Hollingsworth, and J. Carver. Combining self-reported and automatic data to improve programming effort measurement. In *Proc. 10th European Software Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int. Symp. Foundations of Software Eng. (ESEC/FSE 2005)*, pages 356–365, 2005.

[55] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[56] S. D. Huston, J. C. E. Johnson, and U. Syyid. *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming*. Addison, 2003.

[57] M. Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *Proc. Int. Conf. Software Maintenance (ICSM 1995)*, pages 222–229, 1995.

[58] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, 2006.

[59] J. Koenemann and S. P. Robertson. Expert problem solving strategies for program comprehension. In *Proc. SIGCHI Conf. Human Factors in Comput. Syst. (CHI 1991)*, pages 125–130, 1991.

[60] B. Korel and R. Ferguson. Dynamic slicing of distributed programs. *Appl. Math. and Comput. Sci.*, 2(2):199–215, 1992.

[61] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[62] J. Krinke. Static slicing of threaded programs. In *Proc. 1998 ACM SIGPLAN-SIGSOFT Workshop Program Anal. for Software Tools and Eng. (PASTE 1998)*, pages 35–42, 1998.

[63] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proc. 9th European Software Eng. Conf. held jointly with 11th ACM SIGSOFT Int. Symp. Foundations of Software Eng. (ESEC/FSE 2003)*, pages 178–187, 2003.

[64] L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.

[65] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proc. 6th Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC/FSE 2007)*, 2007.

[66] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.

[67] C. H. LeDoux and D. S. Parker, Jr. Saving traces for Ada debugging. In *Proc. 1985 Annu. ACM SIGAda Int. Conf. Ada (SIGAda 1985)*, pages 97–108, 1985.

[68] H. Leroux, A. Réquilé-Romanczuk, and C. Mingins. JACOT: A tool to dynamically visualise the execution of concurrent Java programs. In *Proc. 2nd Int. Conf. Principles and Practice of Programming in Java (PPPJ 2003)*, pages 201–206, 2003.

[69] S. Letovsky. Cognitive processes in program comprehension. *J. Syst. Softw.*, 7(4):325–339, 1987.

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

[71] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[72] L. J. Levrouw, K. M. R. Audenaert, and J. M. V. Campenhout. A new trace and replay system for shared memory programs based on lamport clocks. In *Proc. 2nd Euromicro Workshop Parallel and Distributed Processing*, pages 471–478, 1994.

[73] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *J. Syst. Software*, 7(4):341–355, 1987.

[74] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. 13th Int. Conf. Architectural Support for Programming Languages and Operating Syst. (ASPLOS 2008)*, pages 329–339, 2008.

[75] A. D. Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Proc. 4th Int. Workshop Program Comprehension (WPC 1996)*, pages 9–18, 1996.

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

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

[78] K. Mehner. JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In S. Diehl, editor, *Software Visualization*, volume 2269 of *LNCS*, pages 163–175. Springer, 2002.

[79] K. Mehner and A. Wagner. Visualizing the synchronization of Java-Threads with UML. In *Proc. 2000 IEEE Int. Symp. Visual Languages (VL 2000)*, pages 199–206, 2000.

[80] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. *SIGARCH Comput. Archit. News*, 17(2):78–86, 1989.

[81] B. P. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. In *Proc. ACM SIGPLAN 1988 Conf. Programming Language Design and Implementation (PLDI 1988)*, pages 135–144, 1988.

[82] G. J. Myers. *The Art of Software Testing.* Wiley, 1979.

[83] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proc. 2000 ACM SIGSOFT Int. Symp. Software Testing and Anal. (ISSTA 2000)*, pages 180–190, 2000.

[84] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, 2005.

[85] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proc. 1993 ACM/ONR Workshop Parallel and Distributed Debugging (PADD 1993)*, pages 1–11, 1993.

[86] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proc. 1992 ACM/IEEE Conf. Supercomputing (Supercomputing 1992)*, pages 502–511, 1992.

[87] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[88] R. H. B. Netzer and J. Xu. Adaptive message logging for incremental replay of message-passing programs. In *Proc. 1993 ACM/IEEE Conf. Supercomputing (Supercomputing 1993)*, pages 840–849, 1993.

[89] A. Newell and H. A. Simon. *Human Problem Solving*. Prentice, 1972.

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

[91] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP 2003)*, pages 167–178, 2003.

[92] W. J. Orlikowski and J. J. Baroudi. Studying information technology in organizations: Research approaches and assumptions. *Inform. Syst. Res.*, 2(1):1–28, 1991.

[93] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proc. 1st ACM SIGSOFT/SIGPLAN Software Eng. Symp. Practical Software Development Environments (SDE 1984)*, pages 177–184, 1984.

[94] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[95] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Proc. 1988 ACM SIGPLAN and SIGOPS Workshop Parallel and Distributed Debugging (PADD 1988)*, pages 124–129, 1988.

[96] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychol.*, 19(3):295–341, 1987.

[97] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–979, 1990.

[98] G. Pothier, Éric Tanter, and J. Piquer. Scalable omniscient debugging. In *Proc. 22nd Annual ACM SIGPLAN Conf. Object-Oriented Programming, Syst., Languages, and Applicat. (OOPSLA 2007)*, pages 535–552, 2007.

[99] V. P. Ranganath and J. Hatcliff. Pruning interference and ready dependence for slicing concurrent Java programs. In *Proc. 13th Int. Conf. Compiler Construction (CC 2004)*, pages 39–56, 2004.

[100] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004.

[101] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[102] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison, 2nd edition, 2004.

[103] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proc. ACM SIGPLAN 1996 Conf. Programming Language Design and Implementation (PLDI 1996)*, pages 258–266, 1996.

[104] B. G. Ryder. Constructing the call graph of a program. *IEEE Trans. Softw. Eng.*, 5(3):216–226, 1979.

[105] B. Sandén. Coping with Java threads. *Computer*, 37(4):20–27, 2004.

[106] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[107] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. Wiley, 2000.

[108] D. C. Schmidt and S. D. Huston. *C++ Network Programming: Resolving Complexity Using ACE and Patterns*, volume 1. Addison, 2002.

[109] C. B. Seaman. Qualitative methods. In F. Shull, J. Singer, and D. I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, chapter 2, pages 35–62. Springer, 2008.

[110] S. Siegel. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill, 1956.

[111] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proc. 14th ACM SIGSOFT Int. Symp. Foundations of Software Eng. (FSE 2006)*, pages 23–34, 2006.

[112] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, 2008.

[113] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*, volume 1. MIT P., 2nd edition, 1998.

[114] R. Snodgrass. Monitoring in a software development environment: A relational approach. In *Proc. 1st ACM SIGSOFT/SIGPLAN Software Eng. Symp. Practical Software Development Environments (SDE 1984)*, pages 124–131, 1984.

[115] E. Soloway, B. Adelson, and K. Ehrlich. Knowledge and processes in the comprehension of computer programs. In M. T. Chi, R. Glaser, and M. J. Farr, editors, *The Nature of Expertise*, pages 129–152. Erlbaum, 1988.

[116] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267, 1988.

[117] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Sci. Comput. Program.*, 36(2-3):183–207, 2000.

[118] B. Stroustrup. *The C++ Programming Language*. Addison, 3rd edition, 2000.

[119] K. C. Tai and S. Ahuja. Reproducible testing of communication software. In *Proc. 11th Annu. Int. Computer Software and Applications Conf. (COMPSAC 1987)*, pages 331–337, 1987.

[120] K. C. Tai, R. H. Carver, and E. E. Obaid. Deterministic execution debugging of concurrent Ada programs. In *Proc. 13th Annu. Int. Computer Software and Applications Conf. (COMPSAC 1989)*, pages 102–109, 1989.

[121] F. Tip. A survey of program slicing techniques. *J. Program. Lang.*, 3(3):121–189, 1995.

[122] M. W. van Someren, Y. F. Barnard, and J. A. Sandberg. *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic P., 1994.

[123] I. Vessey. Expertise in debugging computer programs: A process analysis. *Int. J. Man-Mach. Stud.*, 23(5):459–494, 1985.

[124] A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. In *Proc. IEEE 2nd Workshop Program Comprehension (WPC 1993)*, pages 78–86, 1993.

[125] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *Proc. 16th Int. Conf. Software Eng. (ICSE 1994)*, 1994.

[126] A. von Mayrhauser and A. M. Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Trans. Softw. Eng.*, 22(6):424–437, 1996.

[127] A. von Mayrhauser and A. M. Vans. Hypothesis-driven understanding processes during corrective maintenance of large scale software. In *Proc. Int. Conf. Software Maintenance (ICSM 1997)*, pages 12–20, 1997.

[128] C. von Praun and T. R. Gross. Object race detection. In *Proc. 16th ACM SIGPLAN Conf. Object-Oriented Programming, Syst., Languages, and Applicat. (OOPSLA 2001)*, pages 70–82, 2001.

[129] M. Weiser. Program slicing. In *Proc. 5th Int. Conf. Software Eng. (ICSE 1981)*, pages 439–449, 1981.

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

[131] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer, 2000.

[132] S. Xie. *Evaluating and Refining Diagrams that Support the Comprehension of Concurrency and Synchronization*. PhD thesis, Univ. Georgia, Athens, GA, 2008.

[133] S. Xie, E. Kraemer, and R. E. K. Stirewalt. Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. In *Proc. 29th Int. Conf. Software Eng. (ICSE 2007)*, 2007.

[134] 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. 15th IEEE Int. Conf. Program Comprehension (ICPC 2007)*, 2007.

[135] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[136] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proc. 30th Annu. Int. Symp. Computer Architecture (ISCA 2003)*, pages 122–135, 2003.

[137] B. Yoon and O. N. Garcia. A cognitive framework of debugging. In *Proc. 7th Int. Conf. Software Eng. and Knowledge Eng. (SEKE 1995)*, pages 304–311, 1995.

[138] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan, 2006.

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

[140] J. Zhao. Slicing concurrent Java programs. In *Proc. 7th Int. Workshop Program Comprehension (IWPC 1999)*, pages 126–133, 1999.

[141] J. Zhao, J. Cheng, and K. Ushijim. Static slicing of concurrent object-oriented programs. In *Proc. 20th Conf. Comput. Software and Applicat. (COMPSAC 1996)*, pages 312–320, 1996.