Using Program Mutation for the Empirical Assessment of Fault Detection Techniques: A Comparison of Concurrency Testing and Model Checking

by

Jeremy S. Bradbury

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
June 2007

Abstract

As a result of advances in hardware technology such as multi-core processors there is an increased need for concurrent software development. Unfortunately, developing correct concurrent code is more difficult than developing correct sequential code. This difficulty is due in part to the many different, possibly unexpected, executions of the program, and leads to the need for special quality assurance techniques for concurrent programs such as randomized testing and formal state space exploration. This thesis focuses on the complementary relationship between such different state-of-the art quality assurance approaches in an effort to better understand the best bug detection methods for concurrent software. An approach is presented that allows the assessment and comparison of different software quality assurance tools using metrics to measure the effectiveness and efficiency of each technique at finding concurrency bugs. Using program mutation, the assessment method creates a range of faulty versions of a program and then evaluates the ability of various testing and formal analysis tools to detect these faults. The approach is implemented and automated in an experimental mutation analysis framework (*ExMAn*) which allows results to be easily reproducible. A comparison of the IBM concurrency testing tool ConTest and the NASA model checker Java PathFinder is given to demonstrate the approach.

Co-Authorship

Chapters 4 and 5 were published previously in papers co-authored with my supervisors James R. Cordy and Juergen Dingel. Both Chapter 4 and Chapter 5 were published in the proceedings of the 2nd *Workshop on Mutation Analysis (Mutation 2006)* [BCD06a, BCD06b]. In Chapter 6, the description of the controlled experiment comparing ConTest and Java PathFinder with depth-first search was accepted for publication at the 3rd *Workshop on Mutation Analysis (Mutation 2007)* in another paper co-authored with my supervisors [BCD07]. For all three papers I was the primary author and conducted the research under the supervision and in collaboration with James R. Cordy and Juergen Dingel. The concepts and ideas in Chapter 3 were published in an earlier form in the proceedings of the 6th *International ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)* [BCD05].

Acknowledgments

I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Graduate Scholarship (OGS) program for their generous financial support.

I would like to thank my supervisors, Juergen Dingel and Jim Cordy, for all of their guidance in helping me grow as a researcher and complete my Ph.D. dissertation. They have both been excellent mentors who have provided me with two great examples of the type of researcher one should strive to become. I would like to thank the members of my thesis examination committee for their helpful comments and insights: John Hatcliff, Mohammad Zulkernine, Tom Dean and Bob Crawford. I would also like to thank all of my friends and members of the School of Computing who have helped me in one way or another along the way. In particular I would like to thank: Richard Zanibbi, Dean Jin, Michelle Crane, Chris McAloney, Derek Shimozawa and Adrian Thurston.

I would like to thank my parents, Goldie and Gerald, my sister, Pamela, and my grandmothers, Jessie and Gladys, for their love and support. It has taken a long time to achieve this goal and my family has always supported the choices I have made.

Finally, I would like to thank Michelle Cortes for her love and support. She has always been there to listen.

Statement of Originality

I, Jeremy Bradbury, certify that the research work presented in this thesis is my own and was conducted under the supervision of James R. Cordy and Juergen Dingel. All references to the work of other people are properly cited.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

A 2002 study by the National Institute of Standards and Technology estimates that the cost of inadequate software testing on the national economy of the United States is \$59.5 billion or 0.6 percent of the United States' gross domestic product [RTI02][1]. The estimated cost to software developers is \$21.2 billion and the cost to software users is \$38.3 billion. The study suggests that feasible improvements to testing infrastructure would provide a potential cost reduction of \$22.2 billion (37% of the cost from inadequate testing).

The cost of inadequate software testing indicates a need for higher quality software development. More rigorous software quality assurance techniques are needed to ensure that we can be confident about the level of quality of the software we produce. In a recent *IBM Systems Journal*, Ritsko and Bates commented on the cost and importance of testing and analysis [RB02]:

> *"Because the cost of testing and verification can exceed the cost of design and programming, the methodologies, techniques, and tools used for testing are key to efficient development of high quality software."*

[1]The global cost of inadequate software testing has been roughly estimated at \$180 billion a year [Woo06].

Software engineering researchers and practitioners have expressed a variety of perspectives and opinions regarding how to develop better software. The variety of perspectives is evident when one considers the range of software processes in use from the waterfall approach [Roy70] to extreme programming [Bec00]. There is also a variety of views regarding the kinds of software quality assurance techniques that are needed. For example, a number of researchers believe that the answer to improved quality assurance is formal analysis. In a recent article titled "Verified Software: A Grand Challenge", Jones, O'Hearn, and Woodcock state that *"Given the right computer-based tools, the use of formal methods could become widespread and transform software engineering"* [JOW06]. Alternatively, others have advocated improvements to current testing techniques as the more realistic approach to improving the state of software quality.

The primary motivation for this thesis lies in the need for better software and the variety of opinions about the appropriate tools to ensure high quality software. We are interested in better understanding the benefits and drawbacks of using different software quality assurance techniques like formal analysis and testing.

1.2 Problem

Our work focuses primarily on better understanding fault detection techniques for concurrent software. Many approaches to ensuring concurrent code is correct have been proposed including: concurrency testing (e.g., ConTest [EFN+02]), model checking (e.g., Java PathFinder [HP00, VHB+03, JPF], Bandera/Bogor [CDH+00, HDPR02, RDH03]), dynamic analysis (e.g., ConAn [LHS03]) and static analysis (e.g., FindBugs [HP04]). The variety of techniques available leads to an important question:

Under which circumstances is one technique more effective or efficient than another?

We are interested in answering this question primarily for testing and model checking, and understanding how to use these different techniques effectively in combination.

The general problem of comparing different quality assurance tools is challenging. Ideally for a given domain and a given kind of fault there would exist an automatic evaluation technique for comparing any new quality assurance technique with existing techniques and tools. However, the reality is that conducting good empirical studies (e.g., controlled experiments) can be time consuming, tedious and error prone. One has to be extremely careful regarding the example systems used in the comparison, the experimental environment, and the proper usage of the different techniques or tools being compared. Comparing two tools with different intended purposes may lead to an unfair comparison that biases one or both of the techniques. In many areas once a controlled experiment is conducted it is reproduced by other researchers, to independently corroborate the results. Contrast this with software engineering, where many of the experiments published in workshops, conferences and journals are not reproduced and published by other researchers [Hie05]. For example, a recent study on empirical research published at the *International Conference on Software Engineering* found that only 1 out of 44 randomly sampled empirical research papers was an independent evaluation conducted by a third-party [ZMM06]. All of the other papers were examples of self-confirmatory studies – *"...the authors play a large role in the production of the object of study (e.g. developed the tool or a method) and performed the evaluation"* [ZMM06]. Reproducing existing studies is an important part of research and can provide independent validation of the original studies' conclusions or identify potential issues and problems not considered by the original researchers. Our work examines solving the above problems with respect to the comparison of fault detection approaches for concurrent source code. We have developed a methodology and framework to automatically conduct experiments that is intended to ease the reproducible of the experiments by others.

1.3 Thesis Statement and Scope of Research

> *Thesis Statement:* For concurrent applications, a mutation analysis approach will allow for a meaningful assessment of the fault detection capabilities of testing and formal analysis.

In the above statement the term mutation analysis is *"...the process of modifying syntactic software artifacts"* [OAL06]. The term fault refers to *"An incorrect step, process, or data definition in a computer program"* [IEE02]. The terms testing and formal analysis are also defined in the standard way. Testing is *"The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component"* [IEE02]. Formal analysis is *"...a mathematically well-founded automated technique for reasoning about the semantics of software with respect to a precise specification of intended behavior for which the sources of unsoundness are defined"* [DHP+07]. An example of a formal analysis technique is model checking that assesses the correctness of a temporal logic specification or assertion with respect to a model of a system or component.

The term assessment refers to the statistical evaluation of testing and formal analysis using mutation measurements to assess the effectiveness and efficiency of each technique. The goal of such an assessment is to answer the following kinds of questions:

- "Is testing tool A better or worse at finding faults than model checker B?"
- "Is testing tool A more efficient at finding faults than model checker B?"

The scope of the research will be restricted to concurrent Java applications. The kind of concurrent Java applications under consideration are multi-threaded programs that have shared variables. We selected this kind of application for two reasons:

1. *Fault Detection Tool Maturity:* existing testing and model checking tools have been used in this domain for a number of years.

2. *Capabilities of Existing Bug Detection Tools Allow for a Symmetric Comparison:* the use of testing and model checking can detect most of the kinds of faults typically found in concurrent applications including deadlock and race conditions.

A secondary motivation for selecting concurrent applications is the lack of research on using mutation with concurrency. Traditionally, mutation has been applied strictly to sequential source code.

1.4 Contributions

The contributions of this thesis are primarily in the areas of software quality assurance, empirical software engineering, and testing and model checking of concurrent applications. Specific contributions include:

- The development of a generalized mutation-based methodology for conducting controlled experiments comparing different quality assurance approaches (e.g., testing, model checking, static analysis) with respect to fault detection capabilities.
- The implementation of the *ExMAn* (EXperimental Mutation ANalysis) framework to automate and support our methodology. The contribution of the *ExMAn* framework includes its abilities to act as an enabler for further research and future experiments.
- The development and implementation of the *ConMAn* (CONcurrency Mutation ANalysis) operators for applying our methodology with concurrent Java applications. These operators have been mapped to an existing concurrency bug pattern taxonomy and are, in that sense, representative of real programmer faults.
- The application of the *ConMAn* operators to programs from an existing concurrency benchmark provides the community with a large set of new programs to use in evaluating fault detection tools for concurrent Java applications.

- Empirical results on the effectiveness and efficiency of testing with ConTest and model checking with Java PathFinder with respect to fault detection for concurrent Java applications.

1.5 Organization of Thesis

In this chapter we have presented and motivated the primary research problem of comparing testing and model checking, the thesis statement, and the contributions of the thesis research. The remaining chapters of the thesis are organized as follows:

- *Chapter 2:* overview of the systems of interest (i.e., concurrent Java software), fault detection techniques for concurrency, program mutation and empirical software engineering. Our discussion of empirical software engineering will include field studies, controlled experiments, benchmarks and case studies.

- *Chapter 3:* overview of the experimental mutation analysis methodology. Specifically, we outline the experimental setup, the experimental procedure, threats to validity and possible outcomes of experiments based on our methods.

- *Chapter 4:* overview of the *ExMAn* framework to support the experimental methodology. We have implemented the *ExMAn* framework in Java and intend to release it publicly.

- *Chapter 5:* overview of the *ConMAn* operators for concurrent Java. These mutation operators are used within the *ExMAn* framework to compare quality assurance techniques for concurrent Java.

- *Chapter 6:* empirical comparison of concurrency testing with ConTest and model checking with Java PathFinder using *ExMAn*, *ConMAn* and the experimental mutation analysis methodology.

- *Chapter 7:* a summary of the thesis, a list of contributions, a discussion of limitations and future work and a statement of conclusions.

Chapter 2

Background

This chapter provides an overview of the background required for the work presented in later chapters of this thesis. We first present an overview of the systems of interest for our research (Section 2.1). That is, we review concurrency mechanisms in Java (J2SE 5.0), an example of a concurrent Java application, and the types of bugs that are typically exhibited in concurrent Java applications. Second, we review existing bug detection techniques, including testing and model checking, that can be used to detect faults in concurrent Java applications (Section 2.2). Third, we summarize related work that uses program mutation (Section 2.3). Recall, that program mutation is used in our approach to compare different fault detection techniques. Fourth, we describe empirical software engineering techniques including field studies, controlled experiments, benchmarks and case studies that are used to better understand fault detection techniques (Section 2.4).

2.1 Systems of Interest

2.1.1 Java Concurrency Mechanisms

While the majority of software systems currently developed in industry are single-threaded sequential programs, there is mounting evidence that *"applications will increasingly need* *to be concurrent if they want to fully exploit CPU throughput gains that have now started becoming available and will continue to materialize over the next several years"* [Sut05]. For example, the Intel Core Duo processor is a dual core processor used in Apple's MacBook Pro and Lenovo Thinkpads. In order to fully exploit this processor, as well as other multicore processors, source code needs to be concurrent. In the past, advances in single processors have led to free speed-up of sequential programs which will no longer occur with multicore technologies.

Many imperative programming languages like Java, which are often used in the development of sequential programs, can also be used for the development of concurrent applications. For example, Java provides a number of synchronization events (e.g., wait, notifyAll) for the development of concurrent programs that can affect the scheduling of threads and access to variables in the shared state [CS98]. The variations in the scheduling of threads means that the execution of concurrent Java programs is non-deterministic. The interleaving space of a concurrent Java program consists of all possible thread schedules [EFBA03].

Threads. Java concurrency is built around the notion of multi-threaded programs. The Java documentation defines a thread as *"...a thread of execution in a program."*[2] A typical thread is created and then started using the start() method and terminates once it has finished running. While a thread is alive it can often alternate between being runnable and not runnable. A number of methods exist that can affect the status of a thread:

- sleep(): will cause the current thread to become not runnable for a certain amount of time.
- yield(): will cause the current thread that is running to pause.
- join(): will cause the caller thread to wait for a target thread to terminate.
- wait(): will cause the caller thread to wait until a condition is satisfied. Another thread notifies the caller that a condition is satisfied using the notify() or notifyAll() method.

[2] `java.lang.Thread` documentation

Synchronization. Prior to J2SE 5.0, Java provided support for concurrent access to shared variables primarily through the use of the synchronized keyword. Java supports both synchronization methods and synchronization blocks. Additionally, synchronization blocks can be used in combination with implicit monitor locks. We will present an example of a synchronized block in Section 2.1.2.

Other Concurrency Mechanisms. In J2SE 5.0, additional mechanisms to support concurrency were added as part of the `java.util.concurrent` library[1]:

- *Explicit Lock (with Condition):* Provides the same semantics as the implicit monitor locks but provides additional functionality such as timeouts during lock acquisition.
  - lock(), lockInterruptibly(), tryLock(): lock acquisition methods.
  - unlock(): lock release method.
  - await(), awaitNanos(), awaitUninterruptibly(), awaitUntil(): will cause a thread to wait (similar to wait() method).
  - signal(), signalAll(): will awaken waiting threads (similar to notify() and notifyAll() methods).
- *Semaphore:* Maintains a set of permits that restrict the number of threads accessing a resource. A semaphore with one permit acts the same as a lock.
  - acquire(), acquireUninterruptibly(), tryAcquire(): permit acquisition methods, some of which block until a permit is available.
  - release(): permit release method that will send a permit back to the semaphore.
- *Latch:* Allows threads from a set to wait until other threads complete a set of operations.

[1]definitions of mechanisms and methods from the `java.util.concurrent` and the `java.util.concurrent.locks` documentation

  - await(): will cause current thread to wait until the latch has finished counting down or until the thread is interrupted.
  - countDown(): will decrement the latch count.

- *Barrier:* A point at which threads from a set wait until all other threads reach the point.
  - await(): used by a set of threads to wait until all other threads in the set have invoked the await() method.
- *Exchanger:* Allows for the exchange of objects between two threads at a given synchronization point.

Built-in Concurrent Data Structures. To reduce the overhead of developing concurrent data structures, J2SE 5.0 provides a number of collection types including ConcurrentHashMap and five different BlockingQueues.

Built-in Thread Pools. J2SE 5.0 also provides a built-in FixedThreadPool and an unbounded CachedThreadPool.

Atomic Variables. The `java.util.concurrent.atomic` package includes a number of atomic variables that can be used in place of synchronization: AtomicInteger, AtomicIntegerArray, AtomicLong, AtomicLongArray, AtomicBoolean, AtomicReference and AtomicReferenceArray. Each atomic variable type contains new methods to support concurrency. For example, AtomicInteger contains atomic methods such as addAndGet() and getAndSet().

2.1.2 Concurrent Programs

To further elaborate on the systems of interest for this thesis we will now provide an overview of an actual concurrent Java application that uses some of the concurrency mechanisms outlined in Section 2.1.1. The example is a modified version of the TicketsOrderSim program, an airline ticketing system simulation, from the IBM Concurrency Benchmark [EU04] (see

```
import java.io.IOException;

public class Main {

    /*
     * First parameter is the number of threads
     * Second parameter is the cushion
     */
    private static int numberThreads = 10;
    private static int cushion = 3;

    public static void main(String[] args) {
        if (args.length < 2){
            System.out.println("ERROR: Expected 2 parameters");
        }
        else {
            numberThreads = Integer.parseInt(args[0]);
            cushion = Integer.parseInt(args[1]);
            new Simulator(numberThreads, cushion);
        }
    }
}

public class Simulator implements Runnable{

    static int Num_Of_Seats_Sold =0;
    int Maximum_Capacity, Num_of_tickets_issued;
    boolean StopSales = false;
    Thread threadArr[] ;
```

```
    public Simulator (int size, int cushion){
        Num_of_tickets_issued = size;
        Maximum_Capacity = Num_of_tickets_issued - cushion;
        threadArr = new Thread[Num_of_tickets_issued];

        /* start the selling of the tickets: create agent threads*/
        for( int i=0; i < Num_of_tickets_issued; i++) {
            System.out.println("Creating thread # " + i);
            threadArr[i] = new Thread (this) ;
            threadArr[i].start();
        }
    }

    /* the selling agent: makes the sale and checks if limit was reached*/
    public void run() {
        synchronized (this) {
            if (StopSales == false) {
                Num_Of_Seats_Sold++;
            }
            if (Num_Of_Seats_Sold == Maximum_Capacity) {
                System.out.println("Over capacity");
                StopSales = true;
            }
        }
        if (Num_Of_Seats_Sold > Maximum_Capacity)
            throw new RuntimeException("bug found - oversold
seats!!");
    }
}
```

Figure 2.1: TicketsOrderSim: A concurrent simulation of multiple agents selling tickets for a flight

Figure 2.1). This example demonstrates the basics of thread creation in Java as well as protecting a critical region with synchronization.

The program has two classes: Main and Simulator. The Main class instantiates the Simulator. The Simulator class has two methods: a constructor and a run method. The constructor has two parameters which define the number of ticket agent threads (size) and the number of excess tickets available (cushion). In other words, there are size agents each selling one ticket and there are actually only size-cushion tickets available. The only task that the constructor is responsible for is creating and starting all of the agent threads.

The run method defines the behavior of each agent thread. The method contains a critical region protected by a synchronized block which ensures that only one agent thread can sell a ticket, update the total number of tickets sold, and set a StopSales flag when the number of available tickets has been sold. An agent thread has to obtain an implicit monitor lock object before executing the critical region. The this monitor lock indicates that the block of code is synchronized on the instance of the Simulator class. In addition to actually selling a ticket, the run method also throws a RuntimeException if the agent threads ever oversell tickets for a flight.

2.1.3 Consequences of Bugs in Concurrent Programs

Bugs that occur in concurrent software can exhibit behaviour and consequences not present in sequential programs. Most of the unique consequences of concurrent programs like deadlock and race conditions occur because of bugs or faults in accessing shared data or controlling access to shared data. Below are the typical consequences of bugs in concurrent programs:

- **Interference (i.e., data race, race condition):** *"...occurs when two or more concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the access from being simultaneous."* [LSW07].
- **Deadlock:**
  - **Deadly Embrace:** *"...a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something in a deadlock cycle. For example, this occurs when a thread holds a lock that another thread desires and vice-versa"* [LSW07].
  - **Deadlock-Blocking:** *"...if a thread depends on another thread to complete for its own completion, then it will deadlock if the other thread never completes"* [LSW07]. For example, if a caller thread is waiting for a target thread to join before proceeding and the target thread is in a deadly embrace with other threads.

- **Livelock:** *"...similar to deadlock in that the program does not make progress. However, in a deadlocked computation there is no possible execution sequence which succeeds, whereas in a livelocked computation there are successful computations, but there are also one or more execution sequences in which no thread makes progress"* [LSW07].
- **Starvation:** *"...An example of starvation is when a thread tries to access a synchronised block and the JVM always gives the lock to some other waiting thread"* [LSW07]. Starvation may occur because of thread priorities.
- **Dormancy:** *"...occurs when a non-runnable thread fails to become runnable"* [LSW07].
- **Incoincidence:** occurs *"...through a call completing at the wrong time (excluding consequences already listed above)"* [LSW07].

As we have shown above, the development of concurrent software offers a new set of challenges not present in the development of sequential code. In addition to the new consequences of concurrency bugs, concurrent software also has an additional difficulty regarding the detection of bugs. A bug that leads to a deadlock or race condition may only occur in a very small number of execution interleavings meaning it is extremely difficult to detect some bugs prior to software deployment.

To explore this further let us consider the TicketsOrderSim example from Section 2.1.2. An example of a fault in this program would be if the synchronized block in the run method only contained the if statement with the condition StopSales == false and not the if statement with the condition Num_Of_Seats_Sold == Maximum Capacity (see Figure 2.2). If this fault was present in the TicketsOrderSim program, some (but not all) of the interleavings would exhibit incorrect behavior. For instance, it would be possible for one agent thread to sell the last available ticket and then a second agent thread to sell an additional ticket ensuring that the StopSales flag does not get set. Other possible interleavings would not exhibit incorrect behavior as long as each agent thread does not get interrupted in between the if

```
/* the selling agent: makes the sale and checks if limit was reached*/
public void run() {
        synchronized (this) {
                if (StopSales == false) {
                        Num_Of_Seats_Sold++;
                }
        }
        if (Num_Of_Seats_Sold == Maximum_Capacity) {
                System.out.println("Over capacity");
                StopSales = true;
        }

        if (Num_Of_Seats_Sold > Maximum_Capacity)
                throw new RuntimeException("bug found - oversold seats!!");
}
```

Figure 2.2: Example fault in `run` method of TicketsOrderSim program

statement that sells a ticket and the if statement that sets the `StopSales` flag.

The difficulty in reasoning about interleavings and detecting concurrency bugs is discussed in [SL05]:

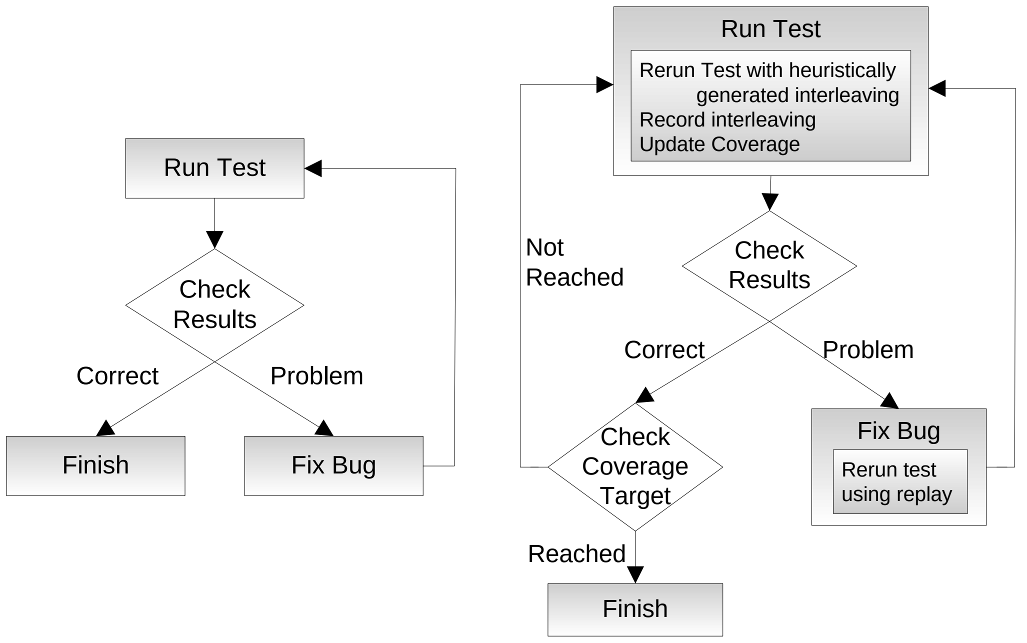
> *"... humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among simple collections of partially ordered operations."*

2.2 Bug Detection Techniques

A variety of quality assurance techniques exist to detect bugs in concurrent programs. Examples of quality assurance techniques include testing and model checking.

2.2.1 Testing

Sequential testing typically involves developing a set of test cases that provide a certain type of code coverage (e.g., path coverage) and executing these tests on the code to detect possible bugs and failures. If a bug is detected in sequential code for a given test case then we can rerun the test case to demonstrate the bug and reuse the test case on a new version



(a) Conventional testing **(b)** Testing with ConTest

Figure 2.3: A comparison of conventional testing and testing with ConTest [EFN+02]

of the software to ensure that the bug no longer occurs (see Figure 2.3(a)).

Due to the non-determinism of the execution of concurrent source code and the high number of possible interleavings, concurrency testing can not rely on coverage metrics alone to guarantee that code is correct. In addition to ensuring that all code is covered we must also provide some probabilistic confidence that bugs that manifest themselves in only a few of the interleavings are found. For example, since a race condition or deadlock may only occur in a small subset of the possible interleaving space, the more interleavings we test the higher our confidence that the bug that caused the race condition or deadlock will be found. Executing a test case once using one possible interleaving provides us the lowest confidence while increasing the number of interleavings executed increases our confidence that bugs will be detected. The current state-of-the-art in concurrency testing is to use the time duration of testing as a measure of confidence [SL05].

Conventional Coverage-Based Testing with Passive Interleaving Exploration. One approach to testing concurrent software is to use conventional sequential techniques with additional measures to enhance the ability of exploring different interleavings. The use of passive interleaving exploration is possible by conducting testing under different conditions that may affect the scheduling of threads. For example, using different operating systems, different versions of language interpreters (e.g., Java Virtual Machines) and different hardware configurations with single and multiple processors can cause different interleavings to occur. The use of different operating systems and different interpreters were used in a recent study comparing fault detection techniques for concurrent software [BDG$^+$04].

Conventional Coverage-Based Testing with Manual Active Interleaving Exploration. An alternative approach to passive exploration would be to use active interleaving exploration in which the source code is hand instrumented with different delays that will affect thread scheduling (e.g., `sleep()`). Alternatively, the code could be instrumented to use thread priorities which will affect scheduling under some operating systems. The use of hand instrumented delays and priorities were also used in the above mentioned comparison of fault detection techniques for concurrent software [BDG$^+$04].

Testing with ConTest [EFN$^+$02]. Although conventional testing as described above can find some faults in source code it may not be capable of finding all faults. For example, hand instrumentation of source code with delays using `sleep()` calls will affect only a subset of the points during execution where different interleavings may occur. A number of these points can not be accessed at the source code level and require instrumentation at a lower level (e.g., Java bytecode). A viable alternative to the passive and active interleaving exploration described above is to use a testing tool like ConTest that provides several benefits including a randomized scheduler and heuristics to increase the confidence that interleavings with a high-risk of bugs are explored (see Figure 2.3(b)). Randomized scheduling is obtained by automatically and systematically inserting random delays into Java bytecode.

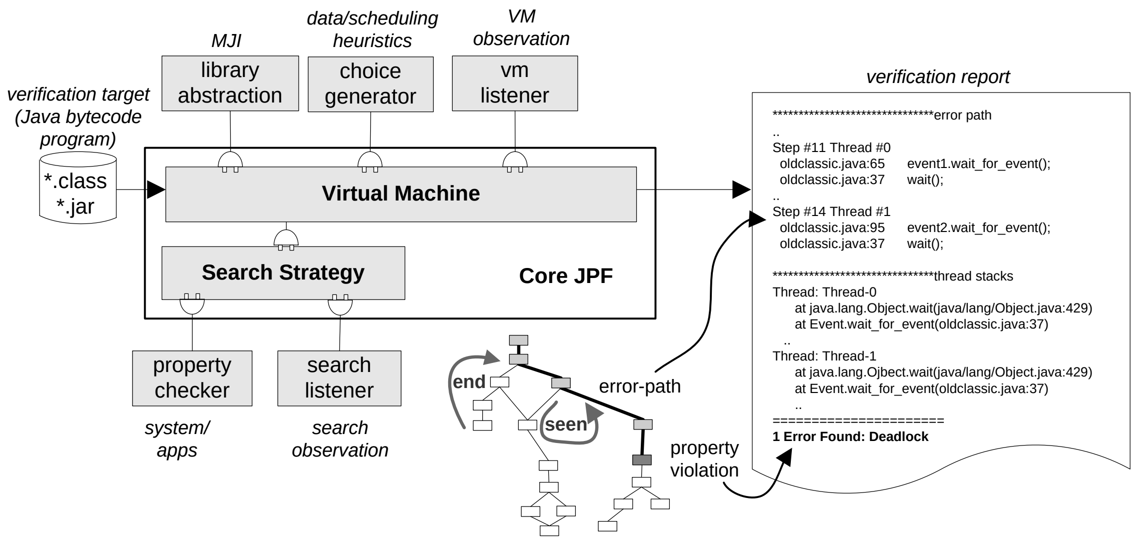
One disadvantage of concurrency testing tools like ConTest is that although random schedulers in concurrency testing allow for the exploration of different interleavings, they do not provide the ability to reproduce or replay a race condition or deadlock. In other words, a test case that produced a bug on one execution is not guaranteed to produce the bug the next time the code is executed. In general, determining if a bug is fixed by rerunning a test case is a non-trivial activity. In order to aid replay in concurrency testing, tools have been developed such as the IBM tool DejaVu [CS98].

2.2.2 Model Checking

Software model checking is a formal methods approach that typically involves developing a finite state model of a software system and specifying a set of assertions or temporal logic properties that the software system should satisfy. The model checker determines if the model of the software system satisfies the specified properties by conducting an exhaustive state space search. The exhaustive search means that all possible interleavings of the model of a concurrent system are examined and thus provides a high level of confidence regarding the quality of the software. Although model checking can provide more confidence then testing it usually requires a long time to search the state space.

Typically model checkers are used to prove correctness, however model checkers also provide benefits as debuggers. Additionally, many software model checkers allow for simulation of the model which can be used like testing to identify possible race conditions or deadlocks. A shift in the focus of techniques like model checking from proofs of correctness to debugging and testing has been advocated by a number of researchers including Rushby [Rus00]. Recent research in formal analysis suggests that this shift is indeed taking place and increases the practicality and acceptance of formal techniques. An example of this shift is demonstrated by the approximately a quarter of a million assertions in the Microsoft Office code [Hoa03]. The primary application of these assertions is *"...not to the proof of program correctness, but to the diagnosis of their errors"* [Hoa03].

Today's state-of-the-art software model checkers are automatic, scalable, and only leave



```
*******************************error path
..
Step #11 Thread #0
  oldclassic.java:65      event1.wait_for_event();
  oldclassic.java:37      wait();
..
Step #14 Thread #1
  oldclassic.java:95      event2.wait_for_event();
  oldclassic.java:37      wait();

*******************************thread stacks
Thread: Thread-0
      at java.lang.Object.wait(java/lang/Object.java:429)
      at Event.wait_for_event(oldclassic.java:37)
   ..
Thread: Thread-1
      at java.lang.Ojbect.wait(java/lang/Object.java:429)
      at Event.wait_for_event(oldclassic.java:37)
      ..
======================
1 Error Found: Deadlock
```

Figure 2.4: Java PathFinder [JPF]

a small semantic gap (if any) between the source artifacts used by developers and the model artifacts required for analysis. The ability of software model checkers to directly analyze source code and the increase in size of systems that can be analyzed has helped them become a viable option for software debugging. For example, in most model checkers a counter-example is produced if the verification of a property fails. When a counter-example is produced it can be used to locate the error in source code. Intuitively, the detection of a property or assertion violation, such as a violation of a method pre-condition, a loop invariant, a class representation invariant, an interface usage rule, or a temporal property should be more insightful than simply knowing that there was a failure of a possibly global test case.

Several software model checkers support the analysis of concurrent Java including Java PathFinder (JPF) [HP00, VHB^{+}03, JPF], developed at NASA, and the Bandera/Bogor tool set [CDH^{+}00, HDPR02, RDH03], developed at Kansas State University. We chose to use JPF for our experiments in this thesis. However, in the future we also plan to conduct further experiments with Bandera/Bogor.

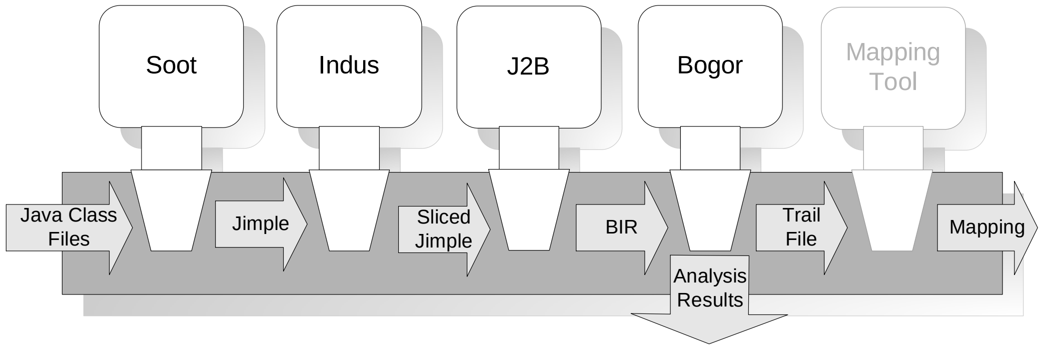


Figure 2.5: The Bandera/Bogor tool set [Ban]

Java PathFinder [HP00, VHB$^+$03, JPF]. Java PathFinder (see Figure 2.4) is an example of an explicit state model checker. Java PathFinder uses Java bytecode as an input language and thus eliminates the semantic gap between source and model artifacts. It also uses a special virtual machine to manage the program state. Java PathFinder is fairly flexible and can use different algorithms to search the state space. For example, you can search the state space of a program in Java PathFinder using a depth first search, breadth first search, heuristic search, or random search. Java PathFinder is also flexible in terms of the type of properties detected. By default it detects deadlocks and exception violations but the user can also create custom properties such as race condition detection. Upon detecting a property violation Java PathFinder will provide both the property that was violated and the error path as output to the user.

Bandera/Bogor [CDH$^+$00, HDPR02, RDH03]. The Bandera/Bogor tool set (see Figure 2.5) supports model checking Java programs by automatically extracting finite-state models from Java source code for model checking. The tool set consists of 5 primary tools:

- *Soot:* Takes Java class files as input and translates them into Jimple, an intermediate representation suitable for optimization.
- *Indus:* Slices the Jimple code.

- *J2B:* Transforms the Jimple code into BIR, the input language for the model checker Bogor.
- *Bogor:* Model checks the BIR models. If a defect in the model is found, a counter example is generated and stored.
- *Mapping Tool:* Maps the counter example back to the Java source code. This tool is currently under development.

Although the combination of Bandera/Bogor is designed for model checking Java programs, the Bogor model checker is itself more flexible and is not tied to any particular programming language. This is the primary difference between Java PathFinder and Bogor – Java PathFinder is language specific (i.e. Java bytecode) and Bogor is not.

2.2.3 Other Techniques

There are a number of other quality assurance techniques that can be used with concurrent software to aide in the detection of faults. For example, static analysis techniques (e.g., JLint [Jli02], FindBugs [HP04]), dynamic analysis techniques (e.g., ConAn [LHS03]) and code inspection can all be used to detect faults and improve the quality of concurrent programs. We will not discuss these other techniques since the remainder of the thesis focuses only on testing and model checking.

2.3 Program Mutation

Program mutation [Ham77, DLS78] is an important part of our research, we use program mutation to create faulty programs (*mutants*) that we can then use to compare the detection capabilities of different fault detection techniques and tools.

A *mutant* is version of the original source code of a program that contains a mutation – *"a single syntactic change that is made to a program statement"* [KO91]. For example:

Original Code:

```
int x=1;
int y=0;

if (x > y ) {
    //print x is bigger!
    ...
} else {
    //print x is not bigger
    ...
}
```

ROR (Relational Operator Replacement):

```
int x=1;
int y=0;

if (x < y ) { // > changed to <
    //print x is bigger!
    ...
} else {
    //print x is not bigger
    ...
}
```

In the above example the only syntactic change is that a "<" is replaced with a ">". The change in the mutant is caused by the application of a *mutant operator* (or pattern). In the case of our example the mutant operator is Relational Operator Replacement (ROR) – one of the traditional mutant operators given in Table 5.1. A variety of other specialized mutant operators exist including the Java inter-class mutant operators [MKO02, OMK04]. Typically, a mutant deviates in behavior from the original program allowing for an analysis of the original and the mutant programs to obtain different results. When a mutant causes an analysis to exhibit a different result we say that the mutant was detected or *killed*. However, sometimes a mutant is functionally equivalent to the original program and is not useful.

An alternative way to define mutation is as a special case of grammar-based testing [OAL06]:

> *"Given a grammar description of a software artifact, mutation operators can be defined to create alternate versions of artifacts. These alternate versions (mutants) can either be valid according to the grammar or invalid. They can be created directly from the grammar or by modifying a ground string (such as a program)."*

Program mutation has a long and rich history – primarily as a coverage technique in the testing community. We will now describe several different uses of mutation including the assessment and coverage of test suites, the assessment of formal analysis and finite state models, the assessment of hybrid techniques and the creation of test data.

2.3.1 Assessment of Testing with Mutation

Test suite assessment is a well developed area and includes such code coverage techniques as: statement coverage, branch coverage, and path coverage. Code coverage techniques are all focused on ensuring a certain level of confidence that a given set of test cases will execute the code in such a way that it will uncover bugs. In mutation coverage a test suite, or any individual test case, is measured by generating a *mutant score – "the percentage of non-equivalent mutants killed by that data"* [OLR$^+$96]. In addition to using mutation to assess test suites, mutation has also been used to assess test suite prioritization techniques [DR05, DR06].

2.3.2 Assessment of Formal Analysis with Mutation

Like test suite assessment, the assessment of a set of properties is often done with respect to coverage. Properties are often assessed with respect to an abstract model of the code (e.g., finite state machines(FSMs)). The use of mutation metrics in formal analysis primarily occurs at the model level not the source code level. For example, mutation metrics have been used to assess state-based coverage of FSMs in model checking [HKHZ99, ABD02].

2.3.3 Assessment of Hybrid Techniques with Mutation

There has been limited uses of mutation to assess hybrid techniques or to compare different quality assurance approaches. One exception is the use of mutation to evaluate the debugging ability of a hybrid approach to assessing concurrent systems. The approach combine testing with ConAn [LHS01], static analysis, and code inspection [LDG$^+$04]. The mutants used in the evaluation of the approach were hand-created and not based on the use of mutation operators.

2.3.4 Creation of Test Data with Mutation

Mutation has been used in the formal methods community to aide in test case generation. Ammann, Black and Majurski use mutation operators to mutate both finite state models and temporal logic properties. A model checker is then used to generate test data that will identify a mutant from the original finite state model [ABM98].

Mutation has also been used in the security engineering community to generate test data for network protocol vulnerability testing [TKD04]. Specifically, mutation was used to create mutant packets and evaluate the ability of the routing protocol OSPF (Open Shortest Path First) to handle the packets.

2.4 Empirical Software Engineering

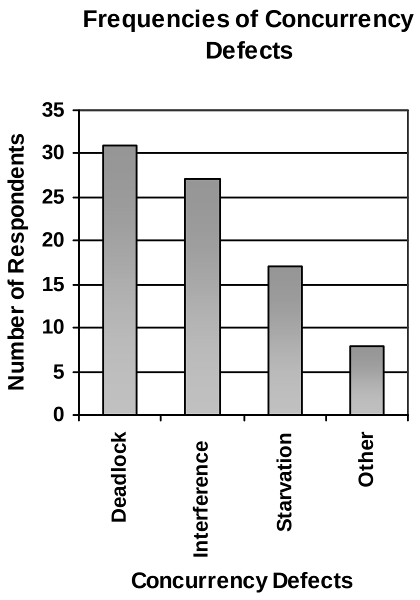
We will now provide an overview of some of the different types of empirical research as well as specific examples of each that are related to defect detection techniques and this thesis. The four research methods we will overview are field studies, controlled experiments, benchmarks and case studies.

2.4.1 Field Studies

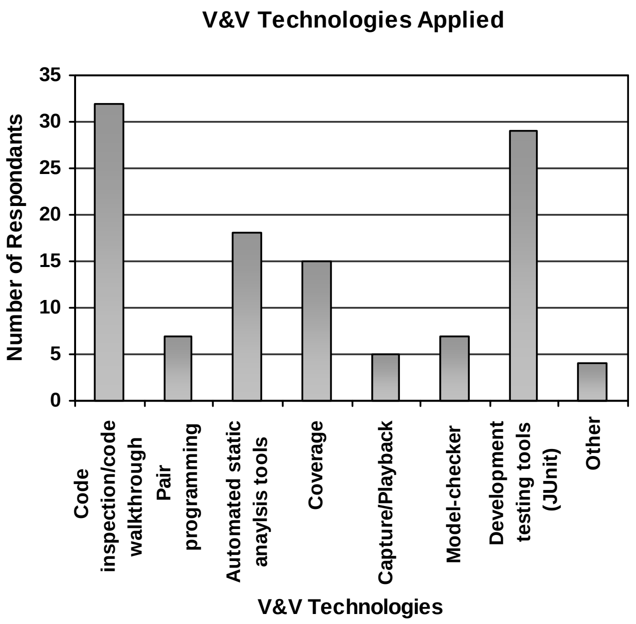
Field studies provide the opportunity to understand observations in both social and organizational contexts [KM99]. Field studies in software engineering are often conducted with several different research goals in mind [LSS05]:

- Identification or derivation of software tool and environment requirements.
- Improvement of practices for software engineering.
- Identification of new hypotheses (theories).

The data analyzed in field studies can be collected through direct involvement of software engineers (e.g., interviews, questionnaires, shadowing), indirect involvement of software



(a) Concurrency defects that respondents have issues with in their work. [WS06]



(b) V&V technologies applied by respondents. [WS06]

Figure 2.6: Results from the Wojcicki and Strooper questionnaire

engineers (e.g., video recordings by participants), and by studying the work artifacts only (e.g., analysis of tool logs, analysis of documentation) [LSS05].

An example of a field study conducted involving direct involvement of software engineers is the questionnaire organized by Wojcicki and Strooper [WS06]. In their research, Wojcicki and Strooper surveyed 35 software practitioners regarding verification and validation (V&V) technique usage for concurrent programs. The majority of respondents were Java developers. Some of the results obtained from the survey include details regarding the types of defects that respondents encounter (see Figure 2.6(a)) and the verification and validation techniques respondents use (see Figure 2.6(b)).

2.4.2 Controlled Experiments

While field studies are often used to identify theories, controlled experiments can be used to confirm or validate theories [WRH$^+$00]. A controlled experiment consists of 6 stages: conception, design, preparation, execution, analysis and dissemination [Pfl95]. Controlled experiments typically involve hypothesis testing in which a null hypothesis (H_0) and an alternative hypothesis (H_A) are defined. In addition to defining an hypothesis the identification of dependent and independent variables is required. Dependent variables are the variables being measured by the experiment. Independent variables are classified as either factors or fixed variables. Factors are the independent variables that the experimenter applies with different treatments. The effect of each treatment is measured with respect to the dependent variables. To ensure that the treatment of factors is responsible for any changes in the dependent variables all other independent variables should be identified and fixed.

Sometimes in software engineering we see quasi-experiments and exemplars used instead of controlled experiments. The difference between a controlled experiment and a quasi-experiment is that a quasi-experiment is a *"Weaker form of (controlled) experiment with a small sample size"* [EA06]. An exemplar is often a weaker form still and is defined as *"...something interesting that was tried on a toy problem"* [EA06].

There have been several empirical studies recently that have used controlled experiments with mutation. For example, Andrews, Briand and Labiche conducted a controlled experiment using a set of sequential C programs to compare the difficulty of using testing to detect mutant bugs in contrast with hand-seeded and real bugs [ABL05]. In this experiment the different types of bugs were the treatments. Controlled experiments have also been used in combination with mutation by Do and Rothermel to compare different test case prioritization techniques [DR05, DR06] (previously mentioned in Section 2.3.1).

There have also been several empirical studies that have focused on the ability of different defect detection tools to find bugs. For example, Rutar, Almazan and Foster conducted a comparison of Bandera, ESC/Java 2, FindBugs, JLint, and PMD on 5 realistic Java applications looking for real faults [RAF04]. The study looked at the type of bugs each tool was capable of detecting. One of the types of bugs included in the study was a concurrency bug called "possible deadlock". All of the static analysis tools (ESC/Java 2, FindBugs, JLint, and PMD) were capable of detecting possible deadlocks. The model checker Bandera 0.3b2, which was a precursor to the current Bandera/Bogor discussed previously in Section 2.2.2, was used as a translation and optimization tool set. It worked with the Spin and Java PathFinder model checkers and was unable to model check the examples used in the study because it could not handle the Java library calls. One of the important conclusions the authors reached after completing their experiment is that *"...we believe there is still a wide area of open research in understanding the right trade-offs to make in bug finding tools"* [RAF04].

Brat el al. conducted a different controlled experiment involving traditional testing, runtime analysis, model checking and static analysis [BDG+04]. The experiment involved human participants using the different techniques to detect 12 seeded faults in NASA's Martian Rover software. The types of faults included deadlock, data races and other non-concurrency faults. Although no statistically significant conclusions were drawn from the experiment the authors stated that the results *"...confirmed our belief that advanced tools can out-perform testing when trying to locate concurrency errors"* [BDG+04].

Finally, Dwyer, Person and Elbaum used controlled experiments with path-sensitive error detection techniques to better understand factors that effect the cost of the techniques [DPE06].

2.4.3 Benchmarks

A benchmark is *"...a test or set of tests used to compare the performance of alternative tools or techniques"* [SEH03]. A benchmark has 3 main parts [SEH03]: a motivating comparison, a task sample and performance measures. There are several examples of benchmarks related to defect detection techniques and tools.

BugBench [LLQ$^+$05] is a general defect detection benchmark of 17 programs with real bugs. Four of the programs in the BugBench benchmark have a concurrency bug. An HTTP server (HTTPD) contains a data race and three different versions of the database MSQL contain a data race and two atomicity bugs.

Another example benchmark is the IBM Concurrency Benchmark [EU04]. The IBM benchmark contains 40 programs ranging in size from 57 to 17000 lines of code. The task samples include student created programs, tool developer programs, open source programs, and a commercial product. One of these programs, the TicketsOrderSim was presented in Section 2.1.2. In Chapter 6 we will use TicketsOrderSim and other example programs from the IBM Concurrency Benchmark to demonstrate our research methods.

2.4.4 Case Studies

Case studies *"...are used primarily for exploratory investigations, both prospectively and retrospectively, that attempt to understand and explain phenomenon or construct a theory. They are generally observational or descriptive in nature..."* [PSE06]. An example of a case study involving testing and model checking was conducted by Chockler, Farchi, Glazberg et al., who compared ConTest and the ExpliSAT model checker within real projects at IBM [CFG$^+$06]. The results of the case study focused on the usage and the comprehensiveness of the results of each tool. Overall, ConTest was found to be easier to use but was not as comprehensive in identifying potential problems in the software. The comprehensiveness considered by Chockler et al. is a similar measurement to our effectiveness.

Chapter 3

An Empirical Methodology for Comparing the Effectiveness and Efficiency of Fault Detection Techniques

In this chapter, we provide an overview of our empirical methodology. A methodology is defined as *"a body of methods, rules, and postulates employed by a discipline : a particular procedure or set of procedures"*[1]. Our methodology is developed with the intention of being able to compare different fault detection techniques based on their effectiveness as well as their efficiency at finding faults. We first motivate the need for a new methodology and approach in Section 3.1. In Section 3.2 we give a high-level overview of the approach included definitions of the metrics collected. We outline the experimental setup methods in Section 3.3 and the experimental procedure in Section 3.4. In our descriptions of both

[1]Merriam-Webster Dictionary

the experimental setup and procedure we define methods and rules for conducting experiments using our methodology. We address possible threats to validity in Section 3.5 and describe the possible outcomes of applying the methodology to testing and model checking in Section 3.6.

3.1 Motivation

Recent work on studying the state of empirical research in software engineering has indicated a need for improved methods and practices in empirical software engineering papers [ZMM06]. Zannier, Melnik and Maurer, compared empirical evaluations published in the *International Conference on Software Engineering (ICSE)* over the past 29 years. The results of their comparison showed that while the number of empirical evaluations has increased in 29 years of ICSE proceedings the authors were unable to conclude that the soundness of empirical evaluations has improved [ZMM06].

The overall goal of our research is to define improved methods for assessing and comparing different fault detection techniques. Specifically, we have developed our methodology to answer the following question from Chapter 1 with respect to fault detection techniques such as testing and model checking:

Under which circumstances is one technique more effective or efficient than another?

Our interest in exploring the relationship between testing and model checking is motivated by a need for improved quality assurance techniques for industrial code – especially concurrent code. Testing has been an effective industrial technique for fault detection of sequential programs. Model checking tools that are now available offer the potential to substantially aid in the debugging of concurrent programs. We are interested in exploring the complementary relationship as well as trade-offs between testing and model checking with respect to fault detection. If the techniques are in fact complementary, we hope to use our methodology and future empirical results to identify ways to integrate model checking and testing to better aid in debugging software. Although we are only discussing the use of our methodology for comparing testing and model checking, the methods and rules outlined in the following sections could also be used to compare other fault detection techniques. In the future we would like to explore the relationships between other techniques and tools. However, applying our methodology to other kinds of comparisons is outside the scope of this thesis.

3.2 Methodology Overview

Our methodology for comparing fault detection techniques is a generalization of mutation testing – a well accepted assessment technique that has been used in the testing community for 30 years. Program mutation is traditionally used to evaluate the effectiveness of test suites. Moreover, mutation provides a comparative technique for assessing and improving multiple test suites. A number of empirical studies (e.g., [ABL05, DR05]) have relied on using mutation as part of the experimental process. Although mutation as a comparative technique has been used primarily within the testing community, it does have application in the broader area of quality assurance and fault detection techniques. Our work is based on the idea that mutation can be used to assess testing (e.g., random testing, concurrent testing with tools such as IBM's ConTest), static analysis (e.g., FindBugs [HP04], Jlint [Jli02], Path Inspector [And04]), model checking (e.g., Java PathFinder [HP00], Bandera/Bogor [RDH03]), and dynamic analysis (e.g., ConAn [LHS03]).

The use of program mutation as a proxy for real program faults has been well researched. Jeff Offutt studied the coupling effect of simple mutant faults with more complicated faults [Off92]. Andrews, Briand and Labiche have studied the relationship between mutant faults and real faults with respect to sequential source code [ABL05]. The previous studies on mutation testing establish a firm basis for the use of mutation in empirical

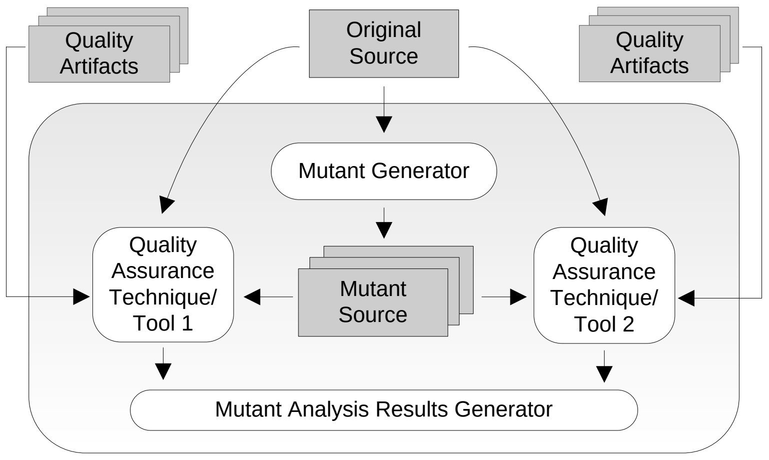

Quality Artifacts
Original Source
Quality Artifacts
Mutant Generator
Quality Assurance Technique/ Tool 1
Mutant Source
Quality Assurance Technique/ Tool 2
Mutant Analysis Results Generator

Figure 3.1: Experimental mutation analysis

research.

The goal of our proposed methodology is to statistically evaluate fault detection techniques using metrics to determine both the quantity of faults found and the efficiency of each approach at finding faults. As mentioned previously, our approach is a generalization of mutation testing and we refer to it as experimental mutation analysis (see Figure 3.1). Experimental mutation analysis has as inputs the original program source along with the quality artifacts for each analysis technique. Examples of quality artifacts would be test cases for testing and assertions or temporal properties for model checking. The first step is to generate mutants from the original program source using a mutation generator. Then for each quality assurance technique or tool we analyze the original program using the quality artifacts to determine the expected output for the program. Next for each assurance technique or tool we analyze each mutant program using the quality artifacts which produces the actual outputs. The mutant analysis results generator then compares the expected and actual outputs and generates the assessment results. We next discuss the metrics that are gathered and produced as the assessment results in Section 3.2.1. We then outline the experimental setup and procedure in our methodology (see Figure 3.2) in Sections 3.3 and 3.4.

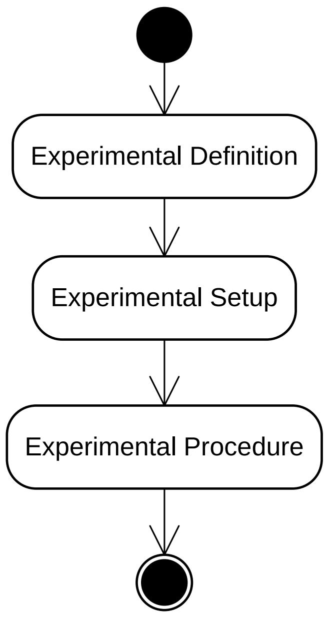


Figure 3.2: High-level experimental methodology activity diagram

Before discussing our metrics, experimental setup and procedure in detail, it is important to clearly explain the role of statistical techniques in our methodology. Statistics allow us to analyze our data and make conclusions and are critical in answering our research questions concerning the effectiveness and efficiency of fault detection technique. However, our methodology does not include a discussion of statistics. Instead, our methodology only outlines how to collect the experimental data not how to analyze the data. We believe that the statistical techniques used will depend on the fault detection techniques being compared, as well as the domain, and should be addressed at the experiment level not the methodology level. The selection of a statistical technique for a given experiment will affect the use of our methodology, in particular our experimental setup methods. For example, depending on the statistical technique used the number of example programs and quality artifacts may vary. We will present an example of using statistics with our methodology in Chapter 6.

3.2.1 Metrics

We collect 4 kinds of metrics in our methodology: mutant score, ability to kill a type of mutant, ease to kill a mutant and cost to kill a mutant. We have chosen these 4 metrics because they provide quantitative measurements pertaining to the effectiveness and efficiency of detecting mutants (proxies for real faults).

To evaluate the effectiveness of a quality assurance technique or tool at killing (detecting) faults we use the mutant score. Recall that we previously described the mutant score metric in Section 2.3.1. The mutant score provides a good comparative measurement to quantify the ability of different fault detection techniques at finding mutant faults.

mutant score of technique t = the percentage of mutants detected (killed) by a technique t (e.g., testing, model checking)

To evaluate the effectiveness of a quality assurance technique or tool at killing a particular kind of fault we measure the ability to kill. We use this metric to help identify any relationships regarding the kinds of faults that are found by a given quality assurance technique. If we consider each mutant operator as generating a different kind of mutant we can measure the ability to detect mutants generated by a given operator.

ability to kill mutants of type s by technique t = the percentage of mutants of type s that are detected (killed) by a technique t .

To evaluate the effectiveness of a quality assurance technique or tool at killing a particular mutant we measure the ease to kill. The ease to kill a mutant is a metric used by Andrews et al. [ABL05].

ease to kill a mutant m by technique t = the percentage of quality artifacts used by technique t that detect (kill) mutant m.

To evaluate the efficiency of a quality assurance technique or tool at killing mutants we measure the cost to kill a mutant. We record both the real time and the CPU time and then can compare techniques based on either the real or CPU time required to detect faults.

cost to kill a mutant m by technique t = the total time (e.g., real or CPU time) required by a technique t to detect (kill) a mutant m.

Although we only use the mutant score, ability to kill, ease to kill and cost to kill metrics, other metrics might be useful to use in our methodology. It might also be appropriate to measure the effectiveness and efficiency in terms of tool specific metrics. For example, in concurrency testing measuring the efficiency as the number of interleavings explored before a mutant is detected.

3.3 Experimental Setup

In this section we define the setup of an experiment based on our methodology. The setup includes rules and guidelines for selecting the approaches under comparison, the example programs used in the experiment, the mutation operators used to generate faults, the quality artifacts used by the approaches under comparison and the experimental environment (see Figure 3.3).

Selection of Approaches for Comparison. When selecting approaches or tools to evaluate it is important to ask two questions:

1. *Are the approaches or tools intended for the same type of applications?* If the tools are specific to different domains or specific to different types of application it may be unfair to compare them if the context of the experiment is outside the scope for which either tool was originally intended. For example, comparing model checking with a conventional testing technique that does not have any mechanism for exploring multiple interleavings for concurrent software would make little sense. Nor would it make sense to compare an analysis technique for embedded systems with one for

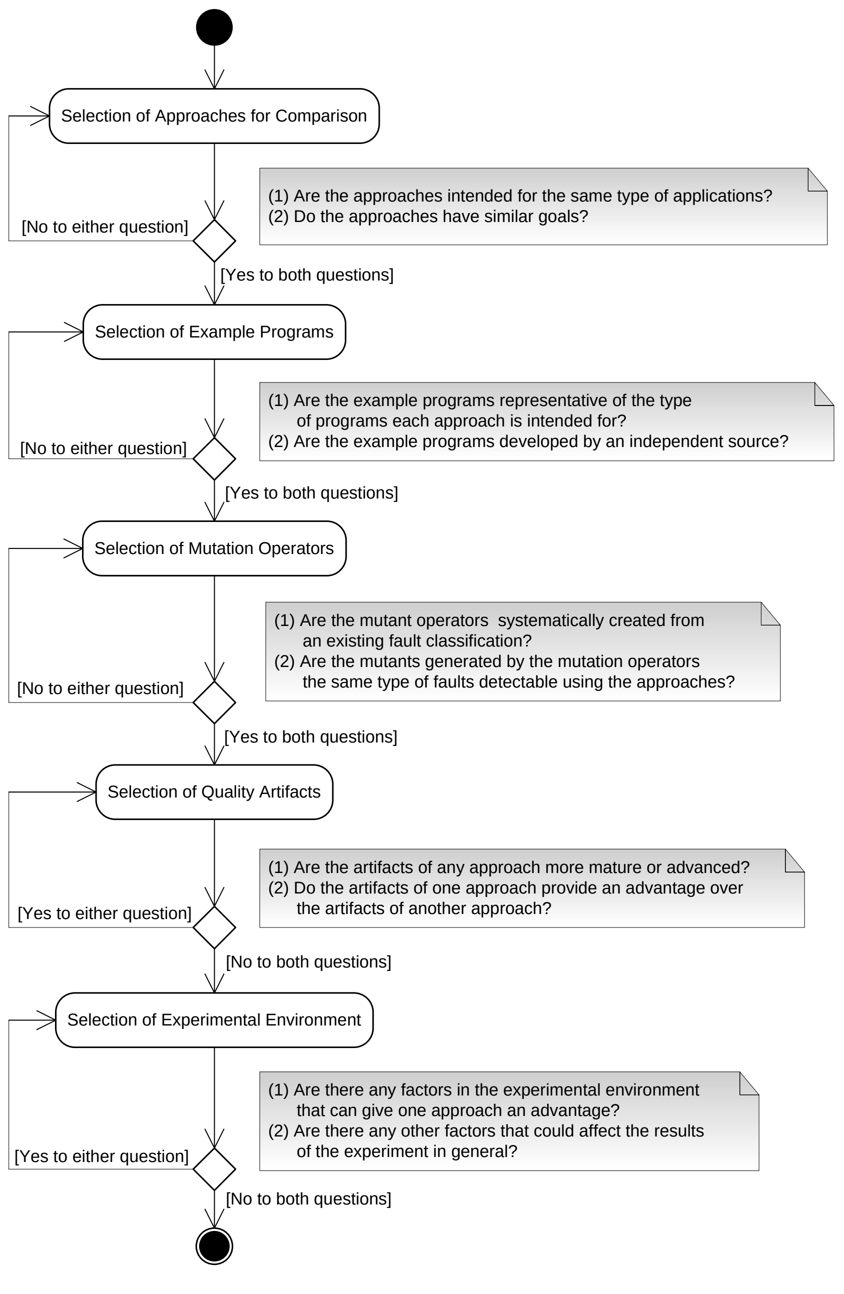

Selection of Approaches for Comparison
(1) Are the approaches intended for the same type of applications?
(2) Do the approaches have similar goals?
[No to either question]
[Yes to both questions]
Selection of Example Programs
(1) Are the example programs representative of the type of programs each approach is intended for?
(2) Are the example programs developed by an independent source?
[No to either question]
[Yes to both questions]
Selection of Mutation Operators
(1) Are the mutant operators systematically created from an existing fault classification?
(2) Are the mutants generated by the mutation operators the same type of faults detectable using the approaches?
[No to either question]
[Yes to both questions]
Selection of Quality Artifacts
(1) Are the artifacts of any approach more mature or advanced?
(2) Do the artifacts of one approach provide an advantage over the artifacts of another approach?
[Yes to either question]
[No to both questions]
Selection of Experimental Environment
(1) Are there any factors in the experimental environment that can give one approach an advantage?
(2) Are there any other factors that could affect the results of the experiment in general?
[Yes to either question]
[No to both questions]

Figure 3.3: Detailed experimental setup activity diagram

The above activity diagram presents the ideal approach to setting up an experiment in our methodology. When applying the experimental setup to real experiments we acknowledge that it may not be possible to answer all of the questions for a given activity before proceeding to the next activity. In cases where question(s) can not be addressed it is up to the discretion of the researcher to assess the effects of not answering the question(s) on the validity of the experiment.

large-scape distributed systems.

2. *Do the approaches or tools have similar goals?* Are they intended to be used for the same purpose and do they find the same types of faults? For example, comparing the fault detection capabilities of two tools where one tool is only capable of detecting deadlocks and the other tool is only capable of detecting race conditions has very little value.

Selection of Example Programs. When selecting example programs for an experiment it is important to ask the following:

1. *Are the example programs representative of the type of programs each approach is intended for?* Given a particular domain are the example programs representative of all of the programs in the domain with respect to purpose, programming constructs, size, and usage. For example, if the approaches are designed for the analysis of reactive systems, then the use of only sequential programs would be inappropriate. An ideal source of representative programs are domain-specific benchmarks.

2. *Are the example programs developed by an independent source?* It is not always possible to get example programs from a third party however every effort should be made to do so and reduce potential bias. A source of independently developed programs is the open source community.

Depending on the approaches under comparison we may be limited in our selection of example programs. One possible limitation may be the size of the example programs due to the scalability of one or more of the approaches. In this case we may have no choice but to choose a set of example programs that are not representative of the type of programs we are interested in. If this occurs we could narrow the scope of our experiment and discuss the selection of example programs as a possible threat to validity in our experiment. We discuss threats to validity in Section 3.5.

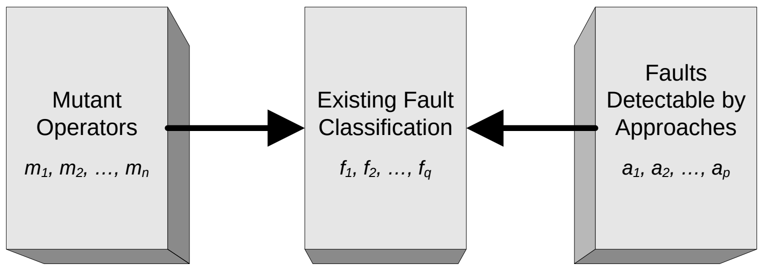


Figure 3.4: The relationship between mutation operators, the approaches under comparison and the fault model

Selection of Mutation Operators. When creating or selecting mutation operators to generate faulty program versions we need to ask two questions:

1. *Are the mutation operators systematically created from an existing fault classification?* A fault classification identifies the kinds of faults that occur in a particular domain. It is important that mutation operators are based on an existing classification to ensure that operators are representative of real programmer faults. Creating the mutation operators in a systematic approach ensures the operators are comprehensive. In Chapter 5 we will present our concurrency mutation operators which are based on an existing bug pattern taxonomy for Java concurrency [FNU03]. We will also discuss related operators including the class-level mutation operators [MKO02] which are based on a fault model of subtype inheritance and polymorphism [OAW$^+$01].

2. *Are the mutants generated by the mutation operators the same type of faults detectable using the approaches?* In Figure 3.4 we present an abstract representation of the relationship between the mutation operators, the fault classification and the fault detection tools. In order to ensure that the fault detection techniques are at least capable of detecting the mutants generated by the mutation operators we need to ensure that the mutation operators and the fault detection tools map to the same kinds of faults in the classification. For example, if f_1 faults were not detectable by any of the fault detection tools we might want to remove the mutation operator m_2

if it only generates f_1 faults.

The above questions are primarily concerned with ensuring that the mutants generated from the operators appear in the underlying fault model and that the mutants are capable of being detected by the approaches.

Selection of Quality Artifacts. When selecting quality artifacts (e.g., test cases, assertions, temporal properties) we should ensure the following questions are addressed:

1. *Are the artifacts of any approach more mature or advanced?* For example, a comparison of testing using a mature test suite with model checking using only a small number of superficial assertions or properties would bias the experiment towards a favorable outcome for testing when in fact the difference in the two approaches might only be attributable to the more mature quality artifacts.

2. *Do the artifacts of one approach provide an advantage over the artifacts of another approach?* This question refers to any other factors that may unfairly influence the comparison. For example, the artifacts used by the approaches might be equally mature but the artifacts for one approach might cover parts of the code not covered by the artifacts of the other.

One way to ensure no bias in the quality artifacts used by each approach is to use the same artifacts for the different approaches. However, this is not always possible.

Selection of the Experimental Environment. When selecting the experimental environment there are several questions that should be considered:

1. *Are there any factors in the experimental environment that can give one approach an advantage?* For example, running an experiment on a multi-processor system when some of the analysis approaches are not multi-threaded might cause a biased comparison in terms of efficiency and invalidate the measurements.

2. *Are there any other factors that could affect the results of the experiment in general?* This question considers other factors that do not necessarily bias one approach but may invalidate the measurements. For example, if the experiment is being run on a shared system then measuring the efficiency of each approach should be done with respect to CPU time not real time.

3.4 Experimental Procedure

In Section 3.3 we outlined how to setup an experiment using our methodology. We now outline how to compare the fault detection capabilities of different techniques and tools in our experimental procedure (see Figure 3.5). There are three main steps of the experimental procedure which are repeated for each example program:

1. *Mutant generation:* A set of mutation operators are applied to an example program to generate mutant programs. Each mutant is the example program with one syntactic change.
2. *Analysis:* Our approach first analyzes the example program to determine the expected observable output. The expected output could include any output generated by the program or the analysis technique. For example, if the analysis approach was testing we could include the standard command-line output, the standard error produced by any exceptions, and the timing information. After obtaining the expected output we analyze each mutant program and compare the mutant output with the expected output. An example of comparing the output of the mutant with the original program would be to use the `diff` program under Linux. It is possible that before comparing the output it may have to be normalized. For example, the output may have to be sorted with a concurrent example program to account for different interleavings. The *Analysis* process for each technique or tool can be conducted sequentially or concurrently.

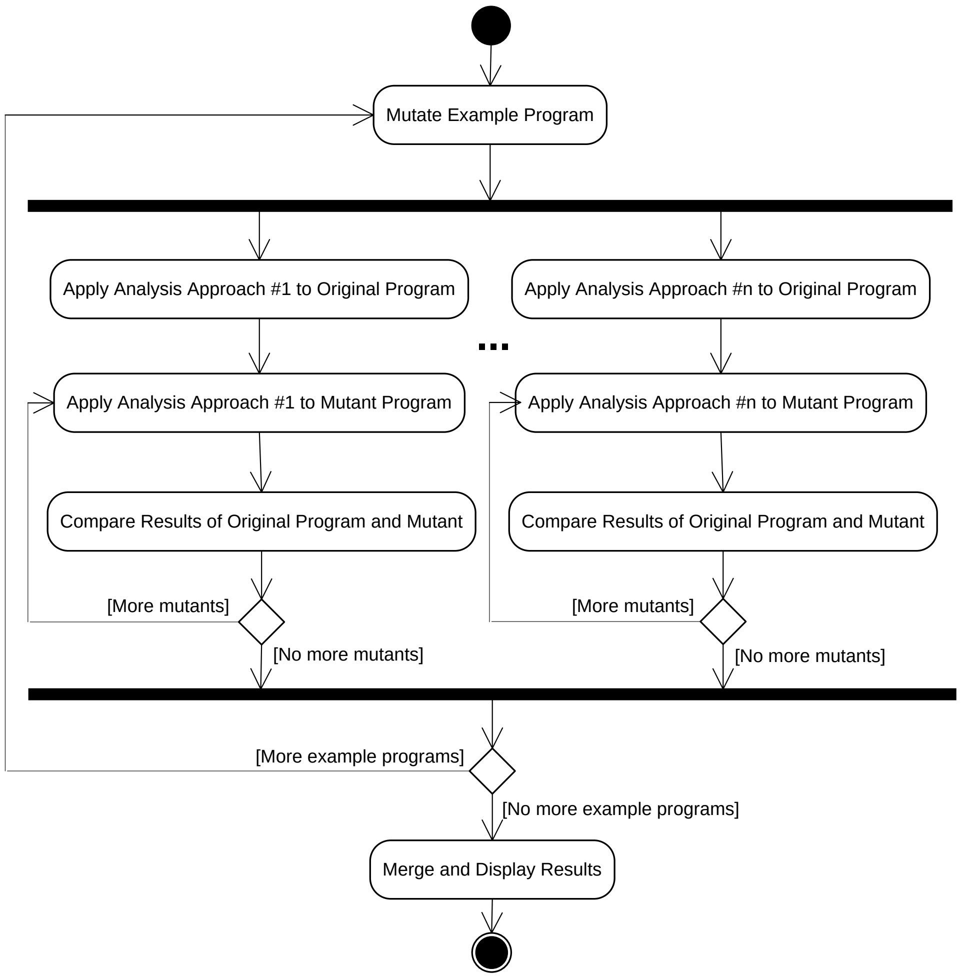

Mutate Example Program
Apply Analysis Approach #1 to Original Program
Apply Analysis Approach #n to Original Program
...
Apply Analysis Approach #1 to Mutant Program
Apply Analysis Approach #n to Mutant Program
Compare Results of Original Program and Mutant
Compare Results of Original Program and Mutant
[More mutants]
[More mutants]
[No more mutants]
[No more mutants]
[More example programs]
[No more example programs]
Merge and Display Results

Figure 3.5: Detailed experimental procedure activity diagram

3. *Merge and display of results:* We compare the analysis results of all of the tools to determine which technique is most effective and efficient. To determine which technique is most effective we compare the mutant scores of each technique and to determine which technique is most efficient we compare the cost to kill a mutant by each technique.

3.5 Threats to Validity

When designing an empirical methodology or a specific experiment there are a number of types of validity that have to be considered: internal validity, external validity, construct validity, and conclusion validity [WRH$^+$00]. We now discuss possible threats to validity in our methodology and how to address them.

Internal validity. Threats to internal validity are *"...influences that can affect the independent variable with respect to causality, without the researcher's knowledge"* [WRH$^+$00]. In our methodology the independent variable is typically the fault detection techniques or tools under comparison. Therefore, threats to internal validity need to be addressed at the experiment level where the specific techniques and tools are known.

External validity. Threats to external validity are *"...conditions that limit our ability to generalize the results of our experiment..."* [WRH$^+$00]. In the type of experiment we describe there are two major threats to external validity. First, a threat to validity is possible if the software being experimented on is not representative of the software to which we want to generalize. We try to address this threat in our experimental setup when we discuss the importance of selecting representative example programs (see Section 3.3). Second, a threat to external validity is possible if the mutant faults used do not adequately represent real faults for the programs under experiment. We discuss how to ensure representative mutants by creating mutation operators based on existing fault classifications in Chapter 5.

Construct validity. Threats to construct validity are *"...concerned with the relation* *between theory and practice"* [WRH$^+$00] and *"...refer to the extent to which the experimental setting actually reflects the construct under study"* [WRH$^+$00]. In our methodology there is a potential for threats to construct validity if the fault detection techniques and tools are not used in the way in which they are intended to be used. We discussed this issue briefly in Section 3.3 when we outlined the importance of selecting tools with similar goals that are applied to the same type of applications. If we ensure that the tools have similar goals we can limit the need to modify how the tools are used.

Conclusion validity. Threats to conclusion validity are *"...concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment"* [WRH$^+$00]. Several concerns in our methodology related to conclusion validity are the confidence that our measurements are correct and the statistical tests used. First, we ensure that our measurements are recorded correctly by automating the collection of measurements. In Chapter 4 we will present a framework to automate our experimental methodology. Second, in order to ensure that the statistical tests used to evaluate the measurements allow for correct conclusions we must ensure that none of the statistical test assumptions are violated and that we use a test with high enough power. The statistical test used in different experiments based on our methodology can vary. Therefore, at the methodology level we can not account for threats to conclusion validity due to the statistical tests. These threats need to be considered at the experiment level when an experiment is designed based on the methods outlined in Sections 3.3 and 3.4.

3.6 Possible Outcomes

Recall that the purpose of experiments based on our methodology is to compare the effectiveness and efficiency of fault detection techniques and tools. We now describe the most probable outcomes of comparing different fault detection techniques and provide specific examples of possible outcomes when comparing testing and model checking. In Chapter 6 we will conduct an actual experiment using the testing tool ConTest and the model checker Java PathFinder.

In terms of effectiveness, there are two outcomes that are most likely:

1. The fault detection techniques are *complementary.* For example, it may be the case that model checking can find bugs that testing can not find and vice versa. In a concurrent setting testing may not be able to find certain bugs at all, while model checking can. In this situation we may be able to identify ways to use the techniques in combination such that the overall effectiveness of detecting faults is increased.

2. The fault detection techniques are *alternatives.* For example, it may be the case that model checking and testing are equally likely to find most of the faults in a concurrent program. In this situation the use of both techniques in combination would provide very little, if any, benefit over either approach in isolation.

Although we outline two distinct outcomes with respect to effectiveness, in reality there is a spectrum of different outcomes. On one end of the spectrum we have completely complementary approaches which do not detect any of the same faults. On the other end of the spectrum we have two completely alternative approaches that detect all of the same faults. In between we have mixed results in which some faults are detected by both approaches and some are not.

In terms of efficiency, there are also a number of possible outcomes:

1. Overall, one fault detection technique is *more efficient.* For example, if the outcome is that testing is always more efficient then model checking with a given tool we would still like to know by what factor is it more efficient because the analysis of the properties might still provide increased insight to developers that can be factored against its increased cost. Therefore, we would also like to know the kinds of properties

that kill mutants and see if they have other benefits for developers in addition to finding faults.

2. A *mixed result* is possible where in certain cases one fault detection technique is more efficient and in other cases a different fault detection technique is more efficient. For example, if a given property kills a number of mutants that are normally killed by a set of test cases, it may be the case that sometimes the execution time for verifying the property is less than the time required to run the test cases. If this is the outcome of our research then an optimal quality assurance approach with respect to efficiency might include both model checking and testing.

3. There is *no statistical difference* between the efficiency of the fault detection techniques. For example, one possibility is that we will find that model checking can be as efficient as testing.

Although there may be other possible outcomes with respect to effectiveness and efficiency, we believe the outcomes listed above are the most likely results.

3.7 Summary

In this chapter we have outlined our empirical methodology for comparing different fault detection techniques like testing and model checking. In particular we have defined the following:

- a general mutation-based approach and effectiveness and efficiency measurements
- methods and rules for experimental setup
- methods and rules for experimental procedure
- potential threats to validity for violations of methodology rules

- potential outcomes of a successful application of the methodology with specific examples regarding the comparison of testing and model checking

Our empirical methodology will be the basis for our experimental mutation analysis framework presented in Chapter 4.

Chapter 4

ExMAn: A Generic and Customizable Framework for Experimental Mutation Analysis

Current mutation analysis tools are primarily used to compare different test suites and are tied to a particular programming language. In this chapter we present the *ExMAn* experimental mutation analysis framework – *ExMAn* is automated, general and flexible and allows for the comparison of different quality assurance techniques such as testing, model checking, and static analysis. *ExMAn* is an implementation of the empirical methodology presented in Chapter 3. The goal of *ExMAn* is to provide a tool that facilitates the mutation-based comparison of different fault detection approaches by:

- providing a maximal degree of automation, which reduces effort and the possibilities for errors
- providing a maximal degree of customization, which allows the tool to be used for a large variety of experiments

- supporting reproducibility of experiments by other researchers

In this chapter we will first provide an overview of the challenges with implementing a mutation analysis framework in Section 4.1. In Section 4.2 we will provide an overview of existing mutation analysis tools that have influenced the design and implementation of *ExMAn*. In Section 4.3 we will provide a description of *ExMAn*'s architecture as well as the functionality of the *ExMAn* framework. In Section 4.4 we outline 7 usage scenarios for *ExMAn*. We will present a summary of the *ExMAn* framework in Section 4.5.

4.1 Challenges with Implementing a Mutation Analysis Framework

In Section 2.3 we introduced program mutation – a technique that has been used in the testing community for 30 years. In Chapter 3 we proposed a generalized methodology called experimental mutation analysis (see Figure 3.1) for comparing the effectiveness and efficiency of different fault detection techniques. Implementing a generalized experimental mutation analysis approach to empirically assess different quality assurance techniques is a challenging problem. A mutation approach that supports the comparison of different quality techniques would have to provide a high degree of automation and customizability. The high degree of automation is required to execute the mutation analysis process and is essential to allow for experimental results to be reproduced. Automation can be achieved through automatically generated scripts to handle the generation of mutants, the mutant analysis, and the generation of results such as mutant score. Customizability is necessary because the approach has to be language and quality artifact independent. On the one hand, language independence means that pluggable mutation generators and compilers are ideal. On the other hand quality artifact independence means the approach should support the comparison of different pluggable quality assurance tools that use artifacts including test cases, assertions, temporal logic properties, and more. In the absence of such a framework, running a wide variety of experiments would mean a considerable amount of effort.

We have developed the *ExMAn* (EXperimental Mutation ANalysis) framework as a realization of our generalized methodology from Chapter 3. That is, *ExMAn* is a reusable implementation for building different customized mutation analysis tools for comparing different quality assurance techniques.

4.2 Related Work: Existing Mutation Tools

There are several mutation tools including Mothra [DGK$^+$88, DO91], Proteum [DM96], and MuJava [OMK04, MOK05] that our work builds upon. The Mothra tool is a mutation tool for Fortran programs that allows for the application of method level mutation operators (e.g., relational operator replacement) (see Table 5.1). The Proteum tool is a mutation analysis tool for C programs. MuJava is the most recent mutation tool and was designed for use with Java and includes a subset of the method-level operators available in Mothra as well as a set of class mutation operators (see Table 5.2) to handle object oriented issues such as polymorphism and inheritance (e.g., mutate the `public` keyword into `protected`).

The difference between *ExMAn* and these tools is that although each is highly automated they were designed to apply mutation analysis to testing. Thus, each is program language dependent and assumes only test cases as quality artifacts. Despite this limitation, all of these tools are excellent for mutation testing and we have learned from their design in building *ExMAn* as a flexible alternative.

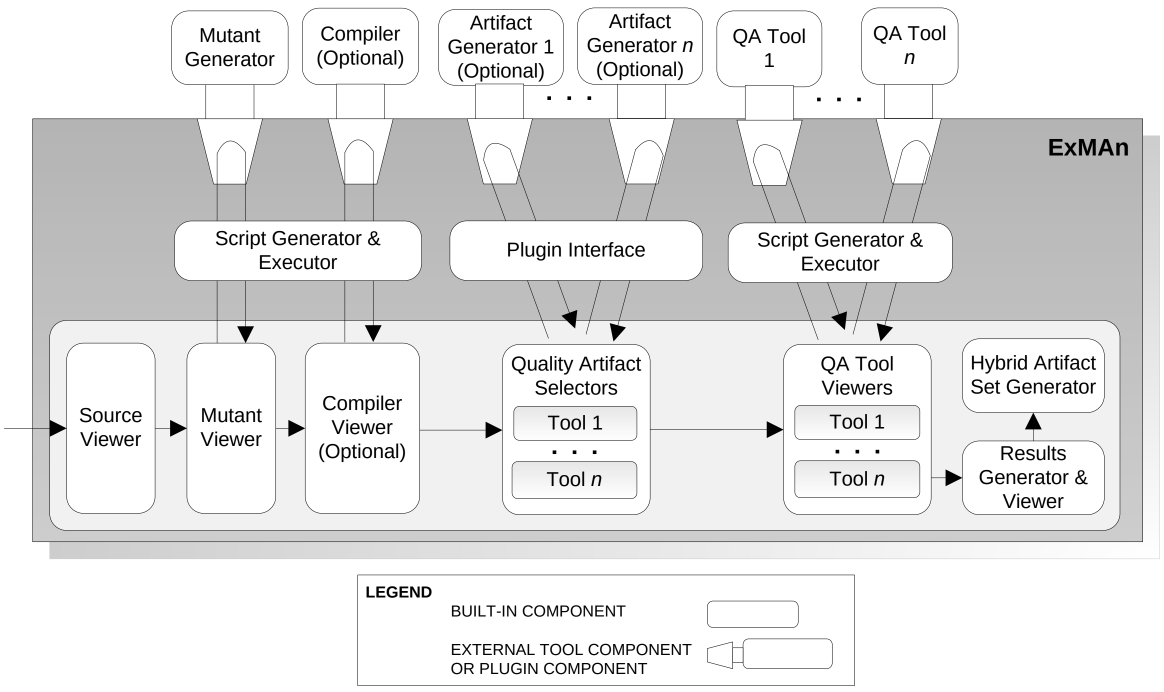

Mutant Generator
Compiler (Optional)
Artifact Generator 1 (Optional)
Artifact Generator *n* (Optional)
QA Tool 1
QA Tool *n*
ExMAn
Script Generator & Executor
Plugin Interface
Script Generator & Executor
Source Viewer
Mutant Viewer
Compiler Viewer (Optional)
Quality Artifact Selectors
Tool 1
Tool *n*
QA Tool Viewers
Tool 1
Tool *n*
Hybrid Artifact Set Generator
Results Generator & Viewer
LEGEND
BUILT-IN COMPONENT
EXTERNAL TOOL COMPONENT OR PLUGIN COMPONENT

Figure 4.1: *ExMAn* architecture

The architecture consists of built-in components (appear inside dark grey box) and external tool components and plugin components (appear outside of grey box at top of diagram). The built-in components in the light grey box provide the ExMAn user interface and allow for control of the external tool components via the Script Generator & Executor. The plugin components are accessed using a plugin interface. Arrows in the diagram represent the typical control flow path between components.

4.3 Overview of *ExMAn*

4.3.1 Architecture

The *ExMAn* architecture is composed of three kinds of components: built-in components, plugin components, and external tool components. The built-in components are general components that are used in all types of experiments (see Figure 4.1). We will discuss most of the general components in our description of the *ExMAn* process in Section 4.3.2. However, we will discuss one important built-in component, the Script Generator & Executor, now. This built-in component provides the interface to the external tool components such as a mutant generator. This component builds and executes scripts when requested by built-in viewer components. Scripts are customized for particular tools based on tool profiles that contain information on the interface of the tool (preferable command line) and a project file that contains information on where input and output are stored. We chose to use a script-based interface for the external tool components because the script interface was more flexible then other interfaces such as a plugin interface and because existing tools are not required to conform to a specific interface.

While the built-in components are general and are used in all mutation analysis experiments, the external components can be replaced, or their usage modified, from one experiment to the next. There are three types of external tool components:

- **Mutant generator.** We can use existing mutant generators such as the Andrews and Zhang C mutant generator tool [AZ03]. We have also designed several custom mutation tools for C and Java using a source transformation language, TXL [Cor06, CDMS02]. We will discuss our custom *ConMAn* mutation operators for concurrent Java in Chapter 5.

- **Compiler.** If we are using a testing approach that requires compiled code we can use standard external compilers such as gcc or javac.

- **Quality Assurance Techniques & Tools.** We can run the mutation analysis on a variety of quality assurance tools including model checkers, static analysis tools, concurrent testing tools, and standard testing techniques.

In addition to the external tool component, there is also one type of plugin component that can be adapted from one experiment to the next:

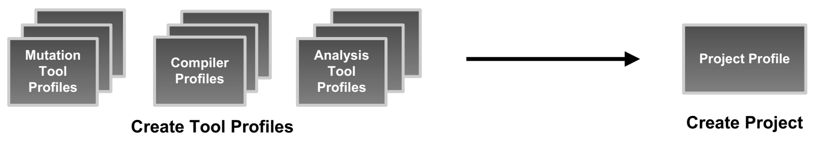
- **Artifact Generator.** We can develop optional customized plugins to generate data for each quality assurance technique in a given experiment. For example, a plugin for testing would produce test cases while a plugin for model checking or static analysis might produce assertions or temporal logic properties.

We have implemented a plugin interface for artifact generators instead of using the script interface because many of the quality assurance tools we are interested in comparing do not have existing artifact generation capabilities. Therefore, we have to create custom generation components instead of using existing external tools. In the future we plan to also provide an alternative script interface for artifact generators to allow us to integrate *ExMAn* with existing test generation tools.

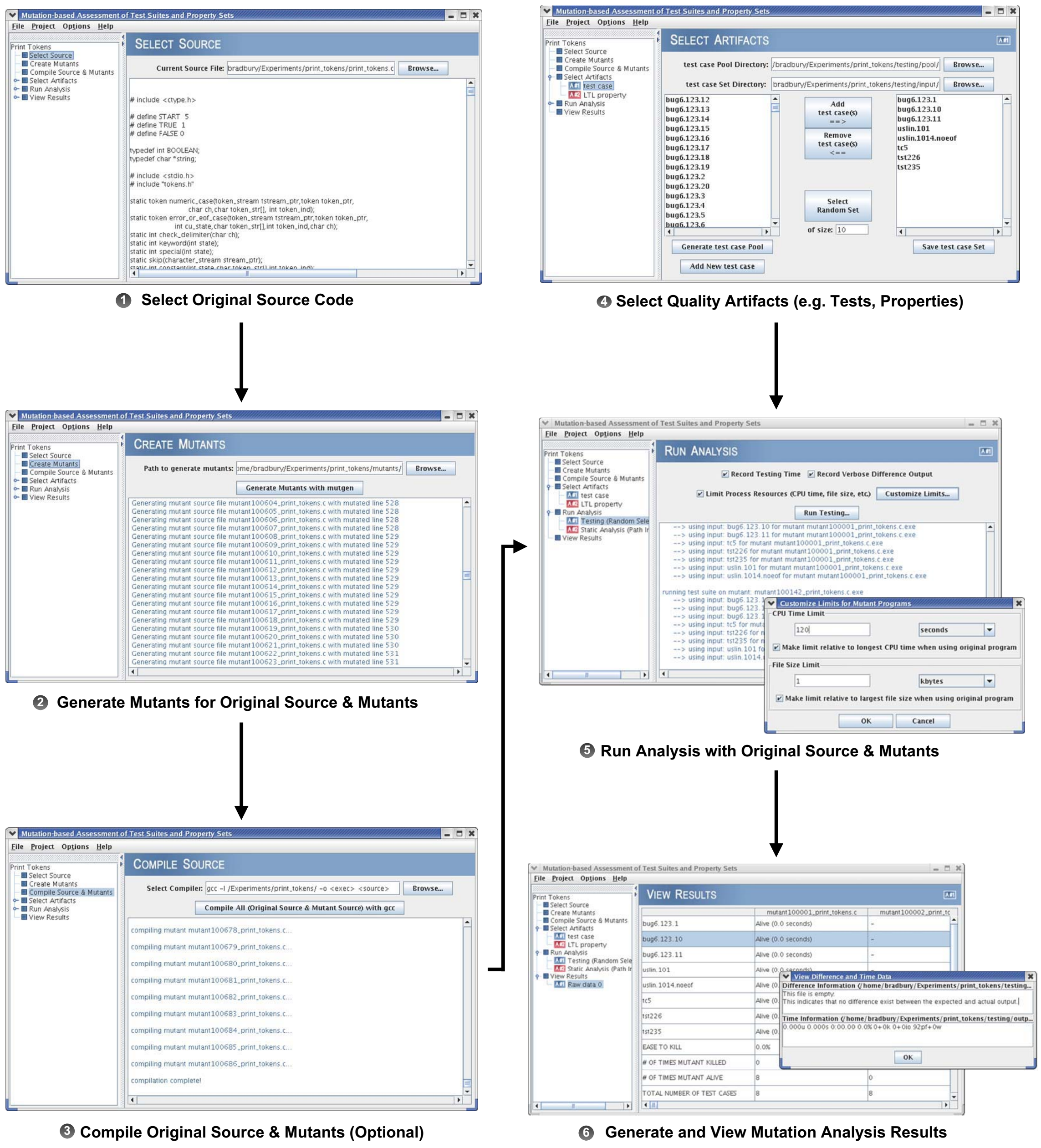
4.3.2 Process Description

Mutation analysis in *ExMAn* requires a setup phase and an execution phase. The setup phase is required because of the generic and customizable nature of the framework (see Figure 4.2(a)). Since *ExMAn* is not tied to any language, or analysis tools, profiles have to be created for using *ExMAn* with specific compilers, mutant generators and analysis tools (see Figure 4.3). A profile contains details on the command-line usage and purpose of the tool. For example to compare concurrent testing using ConTest and model checking using Java PathFinder we would have to ensure that *ExMAn* has defined profiles for a Java compiler, a Java mutant generator (e.g., MuJava) and profiles for executing tests in ConTest as well as model checking with Java PathFinder. *ExMAn* has pre-installed profiles for standard compilers (gcc, javac), mutant generators, and quality assurance approaches (sequential testing, ConTest, Java PathFinder, Bandera/Bogor, Path Inspector). However these tools have to be installed separately and the profiles might have to be edited to include the correct installation paths.

Once tool profiles have been created, a project for a particular experimental mutation analysis has to be defined (see Figure 4.4). The project includes information such as the project name and purpose, the compiler (optional), mutant generator, a finite set of quality assurance analysis tools being compared using mutation, and the paths to all input and output artifacts (e.g., test case input and output directories, mutant directory). When reproducing results a previously created project can be used and the setup phase can be



(a) Setup phase


❶ Select Original Source Code
❷ Generate Mutants for Original Source & Mutants
❸ Compile Original Source & Mutants (Optional)
❹ Select Quality Artifacts (e.g. Tests, Properties)
❺ Run Analysis with Original Source & Mutants
❻ Generate and View Mutation Analysis Results
Mutation-based Assessment of Test Suites and Property Sets
File Project Options Help
Print Tokens
Select Source
Create Mutants
Compile Source & Mutants
Select Artifacts
test case
LTL property
Run Analysis
Testing (Random Sele
Static Analysis (Path Ir
View Results
Raw data 0
Browse...
Generate Mutants with mutgen
Compile All (Original Source & Mutant Source) with gcc
Add test case(s) ==>
Remove test case(s) <==
Select Random Set
Generate test case Pool
Add New test case
Save test case Set
Run Testing...
Customize Limits...
OK
Cancel

SELECT SOURCE

Current Source File: bradbury/Experiments/print_tokens/print_tokens.c

```
# include <ctype.h>

# define START  5
# define TRUE  1
# define FALSE 0

typedef int BOOLEAN;
typedef char *string;

# include <stdio.h>
# include "tokens.h"

static token numeric_case(token_stream tstream_ptr,token token_ptr,
                          char ch,char token_str[], int token_ind);
static token error_or_eof_case(token_stream tstream_ptr,token token_ptr,
                     int cu_state,char token_str[],int token_ind,char ch);
static int check_delimiter(char ch);
static int keyword(int state);
static int special(int state);
static skip(character_stream stream_ptr);
```

CREATE MUTANTS

Path to generate mutants: ome/bradbury/Experiments/print_tokens/mutants/

```
Generating mutant source file mutant100604_print_tokens.c with mutated line 528
Generating mutant source file mutant100605_print_tokens.c with mutated line 528
Generating mutant source file mutant100606_print_tokens.c with mutated line 528
Generating mutant source file mutant100607_print_tokens.c with mutated line 528
Generating mutant source file mutant100608_print_tokens.c with mutated line 529
Generating mutant source file mutant100609_print_tokens.c with mutated line 529
Generating mutant source file mutant100610_print_tokens.c with mutated line 529
Generating mutant source file mutant100611_print_tokens.c with mutated line 529
Generating mutant source file mutant100612_print_tokens.c with mutated line 529
Generating mutant source file mutant100613_print_tokens.c with mutated line 529
Generating mutant source file mutant100614_print_tokens.c with mutated line 529
Generating mutant source file mutant100615_print_tokens.c with mutated line 529
Generating mutant source file mutant100616_print_tokens.c with mutated line 529
Generating mutant source file mutant100617_print_tokens.c with mutated line 529
Generating mutant source file mutant100618_print_tokens.c with mutated line 529
Generating mutant source file mutant100619_print_tokens.c with mutated line 530
Generating mutant source file mutant100620_print_tokens.c with mutated line 530
Generating mutant source file mutant100621_print_tokens.c with mutated line 530
Generating mutant source file mutant100622_print_tokens.c with mutated line 531
Generating mutant source file mutant100623_print_tokens.c with mutated line 531
```

COMPILE SOURCE

Select Compiler: gcc -I /Experiments/print_tokens/ -o <exec> <source>

```
compiling mutant mutant100678_print_tokens.c...

compiling mutant mutant100679_print_tokens.c...

compiling mutant mutant100680_print_tokens.c...

compiling mutant mutant100681_print_tokens.c...

compiling mutant mutant100682_print_tokens.c...

compiling mutant mutant100683_print_tokens.c...

compiling mutant mutant100684_print_tokens.c...

compiling mutant mutant100685_print_tokens.c...

compiling mutant mutant100686_print_tokens.c...

compilation complete!
```

SELECT ARTIFACTS

test case Pool Directory: /bradbury/Experiments/print_tokens/testing/pool/

test case Set Directory: bradbury/Experiments/print_tokens/testing/input/

bug6.123.12	bug6.123.1
bug6.123.13	bug6.123.10
bug6.123.14	bug6.123.11
bug6.123.15	uslin.101
bug6.123.16	uslin.1014.noeof
bug6.123.17	tc5
bug6.123.18	tst226
bug6.123.19	tst235
bug6.123.2	
bug6.123.20	
bug6.123.3	
bug6.123.4	
bug6.123.5	
bug6.123.6	

of size: 10

RUN ANALYSIS

☑ Record Testing Time ☑ Record Verbose Difference Output

☑ Limit Process Resources (CPU time, file size, etc)

```
    --> using input: bug6.123.10 for mutant mutant100001_print_tokens.c.exe
    --> using input: bug6.123.11 for mutant mutant100001_print_tokens.c.exe
    --> using input: tc5 for mutant mutant100001_print_tokens.c.exe
    --> using input: tst226 for mutant mutant100001_print_tokens.c.exe
    --> using input: tst235 for mutant mutant100001_print_tokens.c.exe
    --> using input: uslin.101 for mutant mutant100001_print_tokens.c.exe
    --> using input: uslin.1014.noeof for mutant mutant100001_print_tokens.c.exe

running test suite on mutant: mutant100142_print_tokens.c.exe
    --> using input: bug6.123.1
    --> using input: bug6.123.1
    --> using input: bug6.123.1
    --> using input: tc5 for muta
    --> using input: tst226 for n
    --> using input: tst235 for n
    --> using input: uslin.101 fo
    --> using input: uslin.1014.
```

Customize Limits for Mutant Programs

CPU Time Limit: 120 seconds

☑ Make limit relative to longest CPU time when using original program

File Size Limit: 1 kbytes

☑ Make limit relative to largest file size when using original program

VIEW RESULTS

	mutant100001_print_tokens.c	mutant100002_print_tc
bug6.123.1	Alive (0.0 seconds)	-
bug6.123.10	Alive (0.0 seconds)	-
bug6.123.11	Alive (0.0 seconds)	-
uslin.101	Alive (0.0 seconds)	-
uslin.1014.noeof	Alive (0.	
tc5	Alive (0.	
tst226	Alive (0.	
tst235	Alive (0.	
EASE TO KILL	0.0%	
# OF TIMES MUTANT KILLED	0	
# OF TIMES MUTANT ALIVE	8	0
TOTAL NUMBER OF TEST CASES	8	8

View Difference and Time Data

Difference Information (/home/bradbury/Experiments/print_tokens/testing...

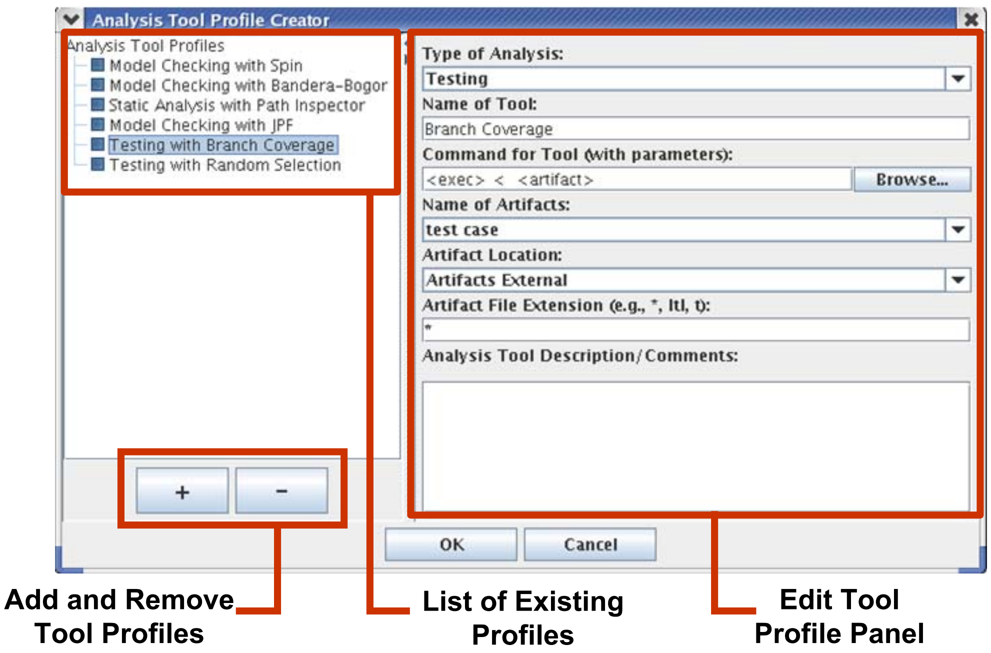
This file is empty.
This indicates that no difference exist between the expected and actual output.

Time Information (/home/bradbury/Experiments/print_tokens/testing/outp...

```
0.000u 0.000s 0:00.00 0.0% 0+0k 0+0io 92pf+0w
```

(b) Execution phase

Figure 4.2: *ExMAn* process


Analysis Tool Profile Creator
+
−
OK
Cancel
Browse...
Add and Remove Tool Profiles
List of Existing Profiles
Edit Tool Profile Panel

Analysis Tool Profiles
- Model Checking with Spin
- Model Checking with Bandera-Bogor
- Static Analysis with Path Inspector
- Model Checking with JPF
- Testing with Branch Coverage
- Testing with Random Selection

Type of Analysis: Testing
Name of Tool: Branch Coverage
Command for Tool (with parameters): <exec> < <artifact>
Name of Artifacts: test case
Artifact Location: Artifacts External
Artifact File Extension (e.g., *, ltl, t): *
Analysis Tool Description/Comments:

Figure 4.3: *ExMAn* Tool Profile Creator dialog

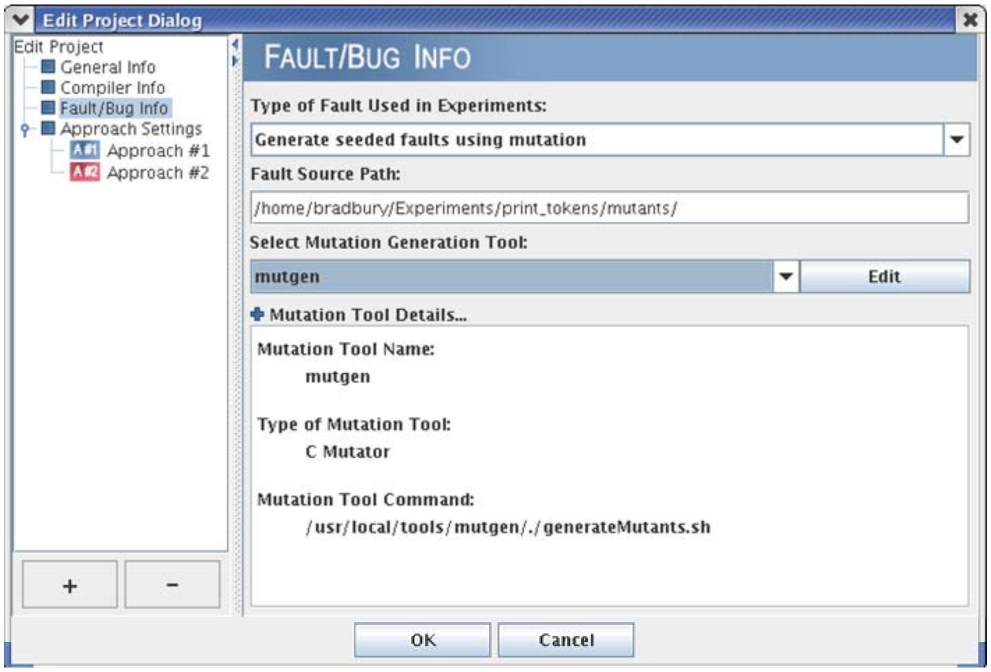
bypassed.

The execution phase occurs once *ExMAn* has been customized for a particular experimental mutation analysis. The execution phase consists of the following steps (see Figure 4.2(b)):

1. *Original Source Code Selection:* select the program or model source to be used in the mutation analysis. A generic language-independent Source Viewer displays the source but does not do any language specific pre-processing to ensure the source is syntactically correct.

2. *Mutant generation:* the mutant generator specified in the project is used to generate a set of mutants for the original source code. The Mutant Viewer reports the progress of the mutant generation.

3. *Compile Original Source Code & Mutants:* an optional step that occurs only if at least



Type of Fault Used in Experiments: Generate seeded faults using mutation

Fault Source Path: /home/bradbury/Experiments/print_tokens/mutants/

Select Mutation Generation Tool: mutgen

Mutation Tool Details...

Mutation Tool Name: mutgen

Type of Mutation Tool: C Mutator

Mutation Tool Command: /usr/local/tools/mutgen/./generateMutants.sh

Figure 4.4: *ExMAn* Create/Edit Project dialog

one of the quality assurance tools (e.g., dynamic analysis, testing, model checking) requires compiled source code as input. The progress of the compilation is reported in the Compile Viewer. Some mutation generation tools produce mutants that are not syntactically correct and thus will not compile. Only mutants that compile will be used in the following steps.

4. *Select Quality Artifacts:* for each quality assurance analysis tool being analyzed using mutation a set of quality artifacts is selected. For example, with model checking a set of temporal logic properties can be selected from a property pool. The property pool can be generated by an optional Artifact Generator plugin or we can use an existing property pool. The selection of quality artifacts can be conducted randomly or by hand using a Quality Artifact Selector & Viewer. For example we could randomly select 20 properties from a property pool or select them by hand. Each quality artifact

can also be viewed in a dialog interface.

5. *Run Analysis with Original Source Code & Mutants:* Quality Analysis Tool Viewers call automatically generated scripts which allow all of the quality assurance tools to be run automatically. For each tool's set of quality artifacts, we first evaluate each artifact using the original source to determine the expected outputs. Next we evaluate the artifacts for all of the mutant versions of the original program. During this step all of the tool analysis results and analysis execution times of each artifact with each program version are recorded and the progress is reported. Quality Analysis Tool Viewers also provide an interface to customize the running of the analysis by placing limits on the size of output and the amount of CPU time. For example, a mutant might cause the original program to go into an infinite loop and never terminate which would be a problem if we are evaluating a test suite. Fortunately, the user can account for this by placing relative or absolute limits on the resources used by the mutant programs. If relative limits are used then the resources used by the original program are recorded and the resources used by each mutant are monitored and the mutant is terminated once it exceeds a relative threshold (e.g. 60 seconds of CPU time more then the original program).

6. *Collection and Display of Results:* results using all of the quality assurance tools are displayed in tabular form in the Results Generator & Viewer. The data presented includes the quality artifact vs. mutant raw data, the mutant score and analysis time for each quality artifact and the ease to kill each mutant (i.e. the number of quality artifacts that kill each mutant). We also can generate hybrid sets of quality artifacts from all quality assurance tools that have undergone mutation analysis using the Hybrid Artifact Set Generator. For instance, if different artifacts are used with different tools we report the combined set of quality artifacts that will achieve the highest mutant score. Additionally, we can generate the hybrid set of artifacts that

achieve a certain mutant score (e.g. 95%) and has the lowest execution cost or smallest set of quality artifacts.

4.4 *ExMAn* in Practice

We will now outline 7 scenarios that demonstrate the flexibility of customizing *ExMAn* for experimental mutation analysis research. The following list is not meant to be comprehensive and other uses in addition to the 7 scenarios are conceivable.

- **Scenario 1: Comparing different test suites.** In Section 3.1 we explained that our methodology is a generalization of mutation testing. We could use *ExMAn* to compare different test suites by using the same testing technique with each test suite.
- **Scenario 2: Comparing different properties in model checking.** If we fix the model checking tool we could compare the ability of different sets of properties to detect faults. This comparison is similar to the comparison of different test suites using a fixed testing technique.
- **Scenario 3: Comparing different test suite prioritization techniques.** In Section 2.3.1 we discussed Do and Rothermel's assessment of test suite prioritization techniques in a controlled experiment using program mutation [DR05, DR06]. We could reproduce their experiment by customizing *ExMAn* to compare testing with one prioritization technique versus testing with another prioritization technique.
- **Scenario 4: Comparing different scheduling techniques in model checking.** In Section 2.4.2 we discussed Dwyer, Person and Elbaum's experimental study of the factors that effect the cost of path-sensitive error detection techniques [DPE06]. One of the factors considered was the search order of the state space in model checking. The experiment conducted by Dwyer, Person and Elbaum used real or hand seed faults instead of mutants. We could reproduce this experiment in *ExMAn* by comparing

Java PathFinder using a depth-first search and fixed scheduling with Java PathFinder using a depth-first search and random scheduling. In this experiment we would use the same quality artifacts (properties and test cases) with both techniques and use program mutation instead of real or hand-seeded faults.

- **Scenario 5: Comparing different search algorithms in model checking.** In the previous scenario we discussed comparing versions of Java PathFinder that have different scheduling techniques. We could also compare versions of Java Pathfinder that have different search algorithms. In Chapter 6 we will conduct a controlled experiment comparing Java PathFinder with depth-first search and Java PathFinder with random search.
- **Scenario 6: Comparing concurrent testing and model checking.** In addition to comparing different versions of the same technique or tool we can also compare two or more completely different fault detection techniques like testing and model checking. For example, in Chapter 6 we will compare testing with ConTest and model checking with Java PathFinder using the *ExMAn* framework. In a comparison of different fault detection techniques we may or may not be able to use the same quality artifacts for all of the techniques.
- **Scenario 7: Comparing sequential testing and static analysis.** In addition to comparing testing and model checking we could also compare other fault detection techniques and tools. For example we could compare sequential testing and static analysis using Path Inspector. Path Inspector allows for the analysis of temporal logic properties.

In addition to demonstrating the customizability of *ExMAn*, the above scenarios also show that our methodology and framework are generalizations of previous research. For example, Scenarios 3 and 4 have been pursued in controlled experiments conducted by other researchers.

4.5 Summary

ExMAn is a generic and flexible framework that allows for the automatic comparison of different quality assurance techniques and the development of combined quality assurance approaches. The flexibility of *ExMAn* occurs because of the separation of the built-in components that can be used in any mutation analysis from the external tool components that place restrictions on the mutation analysis. By using a script invocation interface to access the mutant generator, compiler and quality assurance techniques under analysis we allow them to be easily interchanged with no modifications to the original tools. We have demonstrated the customization of *ExMAn* using one example and are currently planning empirical assessments of testing, static analysis, and formal analysis.

Although *ExMAn* is a generalized and customizable way to conduct mutation analysis it does have limitations and we have identified several areas of future work:

- Add some facility to semi-automatically identify equivalent mutants.
- Add ability to automatically specify patterns for the creation of mutation operators.
- Expand the artifact selection to allow for the selection of multiple quality artifact sets for each type and thus allow for statistical analysis.

We are interested in improving the functionality and flexibility of the *ExMAn* framework and hope to address the above limitations in the near future.

In the next chapter (Chapter 5) we will discuss our concurrency mutation operators before presenting an experiment using *ExMAn* in Chapter 6.

Chapter 5

ConMAn: Mutation Operators for Concurrent Java (J2SE 5.0)

The current version of Java (J2SE 5.0) provides a high level of support for concurrency in comparison to previous versions. For example, programmers using J2SE 5.0 can now achieve synchronization between concurrent threads using explicit locks, semaphores, barriers, latches, or exchangers. Furthermore, built-in concurrent data structures such as hash maps and queues, built-in thread pools, and atomic variables are all at the programmer's disposal (see Section 2.1.1 for more details).

We are interested in using mutation analysis to evaluate, compare and improve quality assurance techniques for concurrent Java programs. Furthermore, we believe that the current set of method mutation operators and class operators proposed in the literature are insufficient to mutate concurrent Java source code because the majority of operators do not directly mutate the portions of code responsible for synchronization. In this chapter we will provide an overview of a new set of concurrent mutation operators – the CONcurrency Mutation ANalysis (*ConMAn*) operators. We will justify the *ConMAn* operators by categorizing them with an existing bug pattern taxonomy for concurrency. Most of the bug patterns in the taxonomy have been used to classify real bugs in a benchmark of concurrent Java applications.

In the next section (Section 5.1) we will motivate the need for a new set of mutation operators specific to concurrent Java and then provide an overview of existing mutation operators for Java (Section 5.2). In Section 5.3 we provide an overview of real concurrency bug patterns which we will use to classify our *ConMAn* operators and demonstrate that the set of operators is both comprehensive and representative of real bugs. The *ConMAn* operators and and the bug pattern classification are presented in Section 5.4. Finally in Section 5.5 we provide a summary and of our work on program mutation and concurrency.

5.1 Motivation

As a result of advances in hardware technology (e.g. multicore processors) a number of practitioners and researchers have advocated the need for concurrent software development [SL05]. Unfortunately, developing correct concurrent code is much more difficult than developing correct sequential code. The difficulty in programming concurrently is due to the many different, possibly unexpected, executions of the program. Reasoning about all possible interleavings in a program and ensuring that interleavings do not contain bugs is non-trivial. Edward A. Lee discussed concurrency bugs in a recent paper [Lee06]:

> *"I conjecture that most multithreaded-general purpose applications are so full of concurrency bugs that - as multicore architectures become commonplace - these bugs will begin to show up as system failures."*

The presence of bugs in concurrent code can have serious consequences including deadlock, starvation, livelock, dormancy, and incoincidence (calls occurring at the wrong time) (see Section 2.1.3) [LSW07].

As we described in Chapter 1, we are interested in using mutation to evaluate, compare, and improve quality assurance techniques for concurrent Java. The use of mutation with

Java has been proposed in previous work – for instance the MuJava tool [MOK05] discussed in Section 4.2. Recall that MuJava included two general types of mutation operators for Java: method level operators [KO91, MOK05] and class level operators [MKO02]. In general, the method and class level mutation operators do not directly mutate the synchronization portions of the source code in Java (J2SE 5.0) that handle concurrency. Furthermore, we conjecture that additional operators are needed in order to provide a more comprehensive set of operators that can truly reflect the types of bugs that often occur in concurrent programs. In this chapter we present a set of concurrent operators for Java (J2SE 5.0). We believe our new set of concurrency mutation operators used in conjunction with existing method and class level operators provide a more comprehensive set of mutation metrics for the comparison and improvement of quality assurance testing and analysis for concurrency.

5.2 Related Work: Existing Mutation Operators for Java

Currently, there are two main groups of operators for mutating Java source code: method and class mutation operators. As previously stated we believe the existing operators are complementary to our concurrency mutation operators.

5.2.1 Method Mutation Operators

Method level operators [KO91, MOK05] have been used in previous mutation tools for other programming languages besides Java (e.g., the Mothra tool set for mutating Fortran programs [DGK$^+$88]). These operators are applied to statements, operands and operators (see Table 5.1). Operators applied to statements perform actions such as modification, replacement, and deletion. Operators applied to operands primarily are replacements. Operators applied to operators include insertion, deletion, and replacement. The sufficient set of method level mutation operators in Table 5.1 have been implemented in the MuJava

	Operator	Description
Method Level Mutation Operators (sufficient set)	AOR	*Arithmetic Operator Replacement*
	AOI	*Arithmetic Operator Insertion*
	AOD	*Arithmetic Operator Deletion*
	ROR	*Relational Operator Replacement*
	COR	*Conditional Operator Replacement*
	COI	*Conditional Operator Insertion*
	COD	*Conditional Operator Deletion*
	SOR	*Shift Operator Replacement*
	LOR	*Logical Operator Replacement*
	LOI	*Logical Operator Insertion*
	LOD	*Logical Operator Deletion*
	ASR	*Assignment Operator Replacement*
Method Level Mutation Operators (not part of sufficient set)	AAR	*Array Reference for Array Reference Replacement*
	ABS	*Absolute Value Insertion*
	ACR	*Array Reference for Constant Replacement*
	ASR	*Array Reference for Scalar Variable Replacement*
	CAR	*Constant for Array Reference Replacement*
	CNR	*Comparable Array Name Replacement*
	CRP	*Constant Replacement*
	DER	*Do Statement End Replacement*
	DSA	*Data Statement Alterations*
	GLR	*Goto Label Replacement*
	RSR	*Return Statement Replacement*
	SAN	*Statement Analysis*
	SAR	*Scalar Variable for Array Reference Replacement*
	SCR	*Scalar for Constant Replacement*
	SDL	*Statement Deletion*
	SRC	*Source Constant Replacement*
	SVR	*Scalar Variable Replacement*

Table 5.1: Method mutation operators [KO91, OLR+96]

	Operator	Description
Class Mutation Operators (Inheritance)	AMC	*Access Modifier Change*
	IHD	*Hiding Variable Deletion*
	IHI	*Hiding Variable Insertion*
	IOD	*Overridding method deletion*
	IOP	*Overridding method calling position change*
	IOR	*Overridding method rename*
	ISI	**`super`** *keyword insertion*
	ISD	**`super`** *keyword deletion*
	IPC	*Explicit call to a parent's constructor deletion*
Class Mutation Operators (Polymorphism)	PNC	*new method call with child class type*
	PMD	*Member variable declaration with parent class type*
	PPD	*Parameter variable declaration with child class type*
	PCI	*Type cast operator insertion*
	PCC	*Cast type change*
	PCD	*Type cast operator deletion*
	PRV	*Reference assignment with other comparable variable*
	OMR	*Overloading method contents replace*
	OMD	*Overloading method deletion*
	OAC	*Arguments of overloading method call change*
Class Mutation Operator (Java-Specific Features)	JTI	**`this`** *keyword insertion*
	JTD	**`this`** *keyword deletion*
	JSI	**`static`** *modifier insertion*
	JSD	**`static`** *modifier deletion*
	JID	*Member variable intialization deletion*
	JDC	*Java-supported default constructor creation*
	EOA	*Reference assignment and content assignment replacement*
	EOC	*Reference comparison and content comparison replacement*
	EAM	*Accessor method change*
	EMM	*Modifier method change*

Table 5.2: Class mutation operators used in MuJava [MKO02]

tool [OMK04, MOK05]. The sufficient set of operators was defined with respect to an empirical study of method operators with a set of Fortran programs [OLR$^+$96]. Further research is also empirically studying the identification of sufficient sets of mutation operators [NA06].

5.2.2 Class Mutation Operators

A set of class level operators [MKO02] were developed and implemented in the MuJava tool [OMK04, MOK05] and are used for the creation of mutants that approximate object oriented bugs. The class level operators are related to inheritance, polymorphism, and Java-specific object oriented features (see Table 5.2). Operators related to inheritance include access modifier changes, overriding method changes, and changes in reference to parent classes. Operators related to polymorphism include changes to class type, type casting, and overloaded methods. Operators related to Java-specific features including inserting and deleting the `this` and `static` keywords.

5.3 Bug Patterns for Java Concurrency

Farchi, Nir, and Ur have developed a bug pattern taxonomy for Java concurrency [FNU03]. The bug patterns are based on common mistakes programmers make when developing concurrent code in practice. Furthermore, the taxonomy has been expanded and used to classify bugs in an existing public domain concurrency benchmark maintained by IBM Research [EU04]. We introduced this benchmark in Section 2.4.3 during our discussion of benchmarks as an empirical software engineering technique. Recall that the benchmark contains 40 programs ranging in size from 57 to 17000 lines of code. Programs in the benchmark are from a variety of sources including student created programs, tool developer programs, open source programs, and a commercial product. In our attempt to develop a comprehensive set of concurrency mutation operators we will later classify our operators with respect to the bug patterns taxonomy. Since this bug pattern taxonomy was developed prior to J2SE 5.0 we added some additional patterns that occur in concurrency constructs not available at the time the taxonomy was proposed. We distinguish between the original bug patterns(*), the added bug patterns also used in the benchmark classification(**) and new patterns that we are including ($^+$):

- **Nonatomic operations assumed to be atomic bug pattern.*** *"...an operation that "looks" like one operation in one programmer model (e.g., the source code level of the programming language). but actually consists of several unprotected operations at the lower abstraction levels" [FNU03].* In this chapter we also include nonatomic floating point operations** in this pattern.

- **Two-state access bug pattern.*** *"Sometimes a sequence of operations needs to be protected but the programmer wrongly assumes that separately protecting each operation is enough" [FNU03].*

- **Wrong lock or no lock bug pattern.*** *"A code segment is protected by a lock but other threads do not obtain the same lock instance when executing. Either these other threads do not obtain a lock at all or they obtain some lock other than the one used by the code segment" [FNU03].*

- **Double-checked lock bug pattern.*** *"When an object is initialized, the thread local copy of the objects field is initialized but not all object fields are necessarily written to the heap. This might cause the object to be partially initialized while its reference is not null" [FNU03].*

- **The sleep() bug pattern.*** *"The programmer assumes that a child thread should be faster than the parent thread in order that its results be available to the parent thread when it decides to advance. Therefore, the programmer sometimes adds an 'appropriate' sleep() to the parent thread. However, the parent thread may still be* *quicker in some environment. The correct solution would be for the parent thread to use the join() method to explicitly wait for the child thread" [FNU03].*

- **Losing a notify bug pattern.*** *"If a notify() is executed before its corresponding wait(), the notify() has no effect and is "lost" ... the programmer implicitly assumes that the wait() operation will occur before any of the corresponding notify() operations" [FNU03].*

- **A "blocking" critical section bug pattern.*** *"A thread is assumed to eventually return control but it never does" [FNU03].*

- **The orphaned thread bug pattern.*** *"If the master thread terminates abnormally, the remaining threads may continue to run, awaiting more input to the queue and causing the system to hang" [FNU03].*

- **Notify instead of notify all bug pattern.**** If a notify() is executed instead of notifyAll() then threads with some of its corresponding wait() calls will not be notified [LDG$^+$04].

- **The interference bug pattern.**** A pattern in which *"...two or more concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the access from being simultaneous."* [LSW07]. The interference bug pattern can also be generalized from classic data race interference to include high level data races** which deal *"...with accesses to sets of fields which are related and should be accessed atomically" [AHB03].*

- **The deadlock (deadly embrace) bug pattern.**** *"...a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something in a deadlock cycle ... For example, this occurs when a thread holds a lock that another thread desires and vice-versa"* [LSW07].

- **Other missing or nonexistent signals.**$^+$ This pattern generalizes the losing a notify bug pattern to all other signals. For example, at a barrier the `await()` method has to be called by a set number of threads before the program can proceed. If an `await()` from one thread never occurs then all of threads at the barrier may be stuck waiting.
- **Starvation bug pattern.**$^+$ This bug occurs when there is a failure to *"...allocate CPU time to a thread. This may be due to scheduling policies..."* [Lea00]. For example, an unfair lock acquisition scheme might cause a thread never to be scheduled.
- **Resource exhaustion bug pattern.**$^+$ *"A group of threads together hold all of a finite number of resources. One of them needs additional resources but no other thread gives one up"* [Lea00].
- **Incorrect count initialization bug pattern.**$^+$ This pattern occurs when there is an incorrect initialization in a barrier for the number of parties that must be waiting for the barrier to trip, or an incorrect initialization of the number of threads required to complete some action in a latch, or an incorrect initialization of the number of permits in a semaphore.

5.4 Concurrent Mutation Operators

We propose five categories of mutation operators for concurrent Java: modify parameters of concurrent methods, modify the occurrence of concurrency method calls (removing, replacing and exchanging), modify keywords (addition and removal), switch concurrent objects, and modify critical regions (shift, expand, shrink and split). The relationship between these general operator categories and the concurrency mechanisms provided in J2SE 5.0 is presented in Table 5.3 – which demonstrates that the operators provide coverage over the J2SE 5.0 concurrency mechanisms.

Java (J2SE 5.0) Concurrency Mutation Operator Categories	*Threads*	*Synchronization methods*	*Synchronization statements*	*Synchronization with implicit monitor locks*	*Explicit locks*	*Semaphores*	*Barriers*	*Latches*	*Exchangers*	*Built-in concurrent data structures (e.g. queues)*	*Built-in thread pools*	*Atomic variables (e.g. LongInteger)*
Modify Parameters of Concurrent Methods	✓	–	✓	✓	✓	✓	✓	✓	–	–	–	–
Modify the Occurrence of Concurrency Method Calls	✓	–	–	–	✓	✓	✓	✓	–	–	–	✓
Modify Keyword	–	✓	✓	✓	✓	–	–	–	–	–	–	–
Switch Concurrent Objects	–	–	–	–	✓	✓	✓	✓	✓	✓	✓	–
Modify Concurrent Region	–	✓	✓	✓	✓	✓	–	–	–	–	–	–

Table 5.3: The relationship between the new *ConMAn* mutation operators and the concurrency features provided by J2SE 5.0

A complete list of the operators we will be presenting in this section is provided in Table 5.4. The mutant operators are designed specifically to represent mistakes that programmers may make when implementing concurrency. Therefore, many of the operators are specific only to concurrency methods, objects and keywords. We have tried to use context and knowledge about Java concurrency to make the operators as specific as possible in order to make concurrency mutation analysis more feasible by reducing the total number of mutants produced.

Readers familiar with method and class level mutation operators will notice that some of our mutation operators are special cases of existing mutation operators while others are new operators that have not been previously proposed. Other related work from the concurrency bug detection community includes a set of 18 hand-created concurrency mutants [LDG$^+$04] for a previous version of Java that did not contain many of the concurrency mechanisms available in J2SE 5.0. We have compared our comprehensive set of operators with this work and found that our operators in combination with the method and class level operators subsume the manual mutants used in the previous work. The idea of using

Operator Category	Concurrency Mutation Operators for Java (J2SE 5.0)
Modify Parameters of Concurrent Methods	M*X*T – Modify Method-X Time *(wait(), sleep(), join(), and await() method calls)*
	MSP - Modify Synchronized Block Parameter
	ESP - Exchange Synchronized Block Parameters
	MSF - Modify Semaphore Fairness
	M*X*C - Modify Permit Count in Semaphore and Modify Thread Count in Latches and Barriers
	MBR - Modify Barrier Runnable Parameter
Modify the Occurrence of Concurrency Method Calls	RT*X*C – Remove Thread Method-X Call *(wait(), join(), sleep(), yield(), notify(), notifyAll() Methods)*
	RC*X*C – Remove Concurrency Mechanism Method-X Call *(methods in Locks, Semaphores, Latches, Barriers, etc.)*
	RNA - Replace NotifyAll() with Notify()
	RJS - Replace Join() with Sleep()
	ELPA - Exchange Lock/Permit Acquisition
	EAN - Exchange Atomic Call with Non-Atomic
Modify Keyword	ASTK – Add Static Keyword to Method
	RSTK – Remove Static Keyword from Method
	ASK - Add Synchronized Keyword to Method
	RSK - Remove Synchronized Keyword from Method
	RSB - Remove Synchronized Block
	RVK - Remove Volatile Keyword
	RFU - Remove Finally Around Unlock
Switch Concur-rent Objects	R*X*O - Replace One Concurrency Mechanism-X with Another *(Locks, Semaphores, etc.)*
	EELO - Exchange Explicit Lock Objects
Modify Critical Region	SHCR - Shift Critical Region
	SKCR - Shrink Critical Region
	EXCR – Expand Critical Region
	SPCR - Split Critical Region

Table 5.4: The *ConMAn* mutation operators for Java

mutation for concurrency was also suggested by Ghosh who proposed two mutation operators (RSYNCHM and RSYNCHB) for removing the synchronized keyword from methods and removing synchronized blocks [Gho02]. The operators proposed by Ghosh are equivalent to the Remove Synchronized Keyword from Method (RSK) and Remove Synchronized Block (RSB) operators presented later in this chapter.

5.4.1 Modify Parameters of Concurrent Method

These operators involve modifying the parameters of methods for thread and concurrency classes. Some of the method level mutation operators that modify operands are similar to the operators proposed here.

MXT - Modify Method-X Timeout

The MXT operator can be applied to the wait(), sleep(), and join() method calls (introduced in Section 2.1.1) that include an optional timeout parameter. For example, in Java a call to wait() with the optional timeout parameter will cause a thread to no longer be runnable until a condition is satisfied or a timeout has occurred. The MXT replaces the timeout parameter, t, of the wait() method by some appropriately chosen fraction or multiple of t (e.g., $t/2$ and $t * 2$). We could replace the timeout parameter by a variable of an equivalent type. However, since we know that the parameter represents a time value it is just as meaningful to mutate the method to both increase and decrease the time by a factor of 2.

Original Code:

```
long time = 10000;
try{
    wait(time);
} catch ...
```

MXT Mutant for wait():

```
long time = 10000;
try{
    wait(time*2);
    //or replace with time/2
} catch ...
```

The MXT operator with the wait() method is most likely to result in an interference bug or a data race when the time is decreased. The MXT operator with the sleep() and join()

methods is most likely to result in the sleep() bug pattern. For example, in a situation where a sleep() or join() is used by a caller thread to wait for another thread, reducing the time may cause the caller thread to not wait long enough for the other thread to complete.

The MXT operator can also be applied to the optional timeout parameter in await() method calls. Both barriers and latches have an await() method. In barriers the await() method is used to cause a thread to wait until all threads have reached the barrier. In latches the await() method is used by threads to wait until the latch has finished counting down, that is until all operations in a set are complete. For example:

Original Code:

```
CountDownLatch latch1
   = new CountDownLatch(1);
...
long time = 50;
latch1.await(time,
    TimeUnit.MILLISECONDS);
...
```

MXT Mutant for await():

```
CountDownLatch latch1
   = new CountDownLatch(1);
...
long time = 50;
latch1.await(time/2,
    TimeUnit.MILLISECONDS);
//or replace time with time*2
...
```

The MXT operator when applied to an await() method call will most likely result in an interference bug.

MSP - Modify Synchronized Block Parameter

Common parameters for a synchronized block include the this keyword, indicating that synchronization occurs with respect to the instance object of the class, and implicit monitor objects. If the keyword this or an object is used as a parameter for a synchronized block we can replace the parameter by another object or the keyword this. For example:

Original Code:

```
private Object lock1 = new Object();
private Object lock2 = new Object();
....
public void methodA(){
    synchronized(lock1){ ... }
}
...
```

MSP Mutant:

```
private Object lock1
   = new Object();
private Object lock2
   = new Object();
...
public void methodA(){
    synchronized(lock2){ ... }
}
...
```

Another MSP Mutant:

```
private Object lock1
   = new Object();
private Object lock2
   = new Object();
...
public void methodA(){
    synchronized(this){ ... }
}
...
```

The MSP operator will result in the wrong lock bug pattern.

ESP - Exchange Synchronized Block Parameters

If a critical region is guarded by multiple synchronized blocks with implicit monitor locks the ESP operator exchanges two adjacent lock objects. For example:

Original Code:

```
private Object lock1
   = new Object();
private Object lock2
   = new Object();
....
public void methodA(){
    synchronized(lock1){
        synchronized(lock2){ ... }
    }
}
...
public void methodB(){
    synchronized(lock1){
        synchronized(lock2){ ... }
    }
}
...
```

ESP Mutant:

```
private Object lock1
   = new Object();
private Object lock2
   = new Object();
....
public void methodA(){
    //switched lock1 and lock2
    synchronized(lock2){
        synchronized(lock1){ ... }
    }
}
...
public void methodB(){
    synchronized(lock1){
        synchronized(lock2){ ... }
    }
}
...
```

The ESP mutation operator can result in a wrong lock bug because exchanging two adjacent locks will cause the locks to be acquired at incorrect times for incorrect critical regions. The ESP operator can also cause a classic deadlock (via deadly embrace) bug to occur as is the case in the above example.

MSF - Modify Semaphore Fairness

Recall in Section 2.1.1 that a semaphore maintains a set of permits for accessing a resource. In the constructor of a semaphore there is an optional parameter for a boolean fairness setting. When the fairness setting is not used the default fairness value is false which allows for unfair permit acquisition. If the fairness parameter is a constant then the MSF operator is a special case of the Constant Replacement (CRP) method level operator and replaces a true value with false and a false value with true. In the case that a boolean variable is used as a parameter we simply negate it.

A potential consequence of expecting a semaphore to be fair when in fact it is not is that there is a potential for starvation because no guarantees about permit acquisition ordering can be given. In fact, when a semaphore is unfair any thread that invokes the Semaphore's acquire() method to obtain a permit may receive one prior to an already waiting thread - this is known as barging[3].

Original Code:

```
int permits = 10;
private final Semaphore sem
  = new Semaphore (permits, true);
...
```

MSF Mutant:

```
int permits = 10;
private final Semaphore sem
  = new Semaphore (permits, false);
...
```

MXC - Modify Concurrency Mechanism-X Count

The MXC operator is applied to parameters in three of Java's concurrency mechanisms: semaphores, latches, and barriers. A latch allows a set of threads to countdown a set of operations and a barrier allows a set of threads to wait at a point until a number of threads reach that point. The count being modified in semaphores is the set of permits, and in latches and barriers it is the number of threads. We will next provide an example of the MXC operator for semaphores, latches, and barriers.

The constructor of the Semaphore class has a parameter that refers to the maximum number of available permits that are used to limit the number of the threads accessing the shared resource. Access is acquired using the acquire() method and released using the release() method. Both the acquire() and release() method calls have optional count parameters referring to the number of permits being acquired or released. The MXC operator modifies the number of permits, p, in calls to these methods by decrementing (p--) and incrementing (p++) it by 1. For example:

Original Code:

```
int permits = 10;
private final Semaphore sem
  = new Semaphore (permits,true);
...
```

MXC Mutant for a Semaphore:

```
int permits = 10;
private final Semaphore sem
  = new Semaphore (permits--,true);
...
```

A potential bug that can occur from modifying permit counts in Semaphores. In the above

[3]`java.util.concurrent` documentation

example if the total number of permits had been one then decrementing the number of permits by 1 would have lead to a situation where no permits were ever available. Another bug could occur if we increased the number of permits acquired by the acquire() method but did not increase the count in the release() method which could eventually exhaust the resources. In this case we could end up with a blocking critical section bug once all of the permits were held but not released.

Similar to the Semaphore constructor's permit count, the constructor of the concurrent latch class CountDownLatch has a thread count parameter that can also be incremented and decremented. For example:

Original Code:

```
int i = 10;
CountDownLatch latch1
    = new CountDownLatch(i);
...
```

MXC Mutant for a Latch:

```
int i = 10;
CountDownLatch latch1
    = new CountDownLatch(i--);
...
```

The MXC operator can also increment and decrement the thread count parameter in the constructor of the concurrent barrier class CyclicBarrier. For example:

Original Code:

```
int i=10;
CyclicBarrier barrier1
    = new CyclicBarrier(i,
    new Runnable(){
        public void run(){
        }
    });
...
```

MXC Mutant for a Barrier:

```
int i=10;
CyclicBarrier barrier1
    = new CyclicBarrier(i++,
    new Runnable(){
        public void run(){
        }
    });
...
```

A potential bug that can occur from modifying the number of threads in Latches and Barriers is resource exhaustion.

MBR - Modify Barrier Runnable Parameter

The CyclicBarrier constructor has a parameter that is an optional runnable thread that can execute after all the threads complete and reach the barrier. The MBR operator modifies the runnable thread parameter by removing it if it is present. This is a special case of the method level mutation operator, Statement Deletion (SDL). For example:

Original Code:

```
int i=10;
CyclicBarrier barrier1
    = new CyclicBarrier(i,
    new Runnable(){
        public void run(){
        }
    });
...
```

MBR Mutant:

```
int i=10;
CyclicBarrier barrier1
    = new CyclicBarrier(i);
//runnable thread parameter removed
...
```

An example of a bug caused by the MBR operator is missed or nonexistent signals if some signal calls were present in the runnable thread.

5.4.2 Modify the Occurrence of Concurrency Method Calls: Remove, Replace, and Exchange

This class of operators is primarily interested in modifying calls to thread methods and methods of concurrency mechanism classes. Examples of modifications include removal of a method call and replacement or exchange of a method call with a different but similar method call. The operators that remove method calls are special cases of the method level operator: Statement Deletion (SDL).

RT*X*C - Remove Thread Method-X Call

The RT*X*C operator removes calls to the following methods: wait(), join(),sleep(), yield(), notify(), and notifyAll(). Removing the wait() method can cause potential interference, removing the join() and sleep() methods can cause the sleep() bug pattern, and removing the notify() and notifyAll() method calls is an example of losing a notify bug. We will now provide an example of the RT*X*C operator used to remove a wait() method call.

Original Code:

```
try{
    wait();
} catch ...
```

RTXC Mutant for wait():

```
try{
    //removed wait();
} catch ...
```

RCXC - Remove Concurrency Mechanism Method-X Call

The RCXC operator can be applied to the following concurrency mechanisms: locks (lock(), unlock()), condition (signal(), signalAll()), semaphore (acquire(), release()), latch (countDown(), and executor service (e.g., submit()).

Let us consider a specific application of the RCXC operator in a ReentrantLock or a ReentrantReadWriteLock with a call to the unlock() method. The RCXC operator removes this call thus the lock is not released causing an example of a blocking critical section bug. For example:

Original Code:

```
private Lock lock1
   = new ReentrantLock();
...
lock1.lock();
try{
   ...
} finally{
   lock1.unlock();
}
...
```

RCXC Mutant for a Lock:

```
private Lock lock1
   = new ReentrantLock();
...
lock1.lock();
try{
   ...
} finally{
 //removed lock1.unlock();
}
...
```

The RCXC operator can also be used to remove calls to the acquire() and release() methods for a semaphore. On the one hand, if an acquire() call is removed interference may occur. On the other hand, if a release() call is removed a blocking critical section bug might be the result.

Original Code:

```
int permits = 10;
private final Semaphore sem = new Semaphore (permits, true);
...
sem.acquire();
...
sem.release();
...
```

RCXC Mutant for a Semaphore:

```
int permits = 10;
private final Semaphore sem
  = new Semaphore (permits, true);
...
//removed sem.acquire();
...
sem.release();
...
```

Another RCXC Mutant for a Semaphore:

```
int permits = 10;
private final Semaphore sem
  = new Semaphore (permits, true);
...
sem.acquire();
...
//removed sem.release();
...
```

Due to the similar nature of applying the RCXC operator for other concurrency mechanisms we will not provide any additional examples.

RNA - Replace NotifyAll() with Notfiy()

The RNA operator replaces a notifyAll() with a notify() and is an example of the notify instead of notify all bug pattern.

Original Code:

```
...notifyAll();...
```

RNA Mutant:

```
...notify();...
```

RJS - Replace Join() with Sleep()

The RJS operator replaces a join() with a sleep() and is an example of the sleep() bug pattern.

Original Code:

```
...join();...
```

RJS Mutant:

```
...sleep(10000);...
```

ELPA - Exchange Lock/Permit Acquistion

In a semaphore the acquire(), acquireUninterruptibly() and tryAcquire() methods can be used to obtain one or more permits to access a shared resource. The ELPA operator exchanges one method for another which can lead to potential timing changes as well as starvation. For example, an acquire() method will try and obtain one or more permits and will block and wait until the permit or permits become available. If the thread that invoked the acquire() method is interrupted it will no longer continue to block and wait. If the acquire() method invocation is changed to acquireUninterruptibly() it will behave exactly the same except it can no longer be interupted. Thus in situations where the semaphore is unfair or if for other reasons the number of requested permits never becomes available the thread that invoked the acquireUninterruptibly() will stay dormant and wait. If an acquire() method invocation is changed to a tryAcquire() then a permit will be acquired if one is available otherwise the thread will not block and wait. tryAcquire() will acquire a permit or permits unfairly even if the fairness setting is set to fair. Use of tryAcquire() may cause starvation for threads waiting for permits.

Original Code:

```
int permits = 10;
private final Semaphore sem = new Semaphore (permits, true);
...
sem.acquire();
...
```

ELPA Mutant:

```
int permits = 10;
private final Semaphore sem = new Semaphore (permits, true);
...
sem.acquireUninterruptibly();
...
```

Another ELPA Mutant:
```
int permits = 10;
private final Semaphore sem = new Semaphore (permits, true);
...
sem.tryAcquire();
...
```

The ELPA operator can also be applied to the lock(), lockInterruptibly(), tryLock() method calls with locks.

SAN - Switch Atomic Call with Non-Atomic

A call to the getAndSet() method in an atomic variable class is replaced by a call to the get() method and a call to the set() method. The effect of this replacement is that the combined get and set commands are no longer atomic. For example:

Original Code:
```
AtomicInteger int1 = 15;
...
int oldVal = int1.getandSet(40);
...
```

SAN Mutant:
```
AtomicInteger int1 = 15;
...
int oldVal = int1.get();
int1.set(40);
...
```

5.4.3 Modify Keywords: Add and Remove

We consider what happens when we add and remove keywords such as static, synchronized, volatile, and finally.

ASTK - Add Static Keyword to Method

The static keyword used for a synchronized method indicates that the method is synchronized using the class object not the instance object. The ASTK operator adds static to non-static synchronized methods and causes synchronization to occur on the class object instead of the instance object. The ASTK operator is an example of the wrong lock bug pattern.

Original Code:
```
public synchronized void a()
{ ... }
```

ASTK Mutant:
```
public static synchronized void a()
{ ... }
```

RSTK - Remove Static Keyword from Method

The RSTK operator removes static from static synchronized methods and causes synchronization to occur on the instance object instead of the class object. Similar to the ASTK operator, the RSTK operator is an examples of the wrong lock bug pattern.

Original Code:
```
public static synchronized void b()
{ ... }
```

RSTK Mutant:
```
public synchronized void b()
{ ... }
```

ASK - Add Synchronized Keyword to Method

The synchronized keyword is added to a non-synchronized method in a class that has synchronized methods or statements. The ASK operator has the potential to cause a deadlock, for example, if a critical region already exists inside the method.

Original Code:
```
public void aMethod()
{ ... }
```

ASK Mutant:
```
public synchronized void aMethod()
{ ... }
```

RSK - Remove Synchronized Keyword from Method

The synchronized keyword is important in defining concurrent methods and the omission of this keyword is a plausible mistake that a programmer might make when writing concurrent source code. The RSK operator removes the synchronized keyword from a synchronized method and causes a potential no lock bug. For example:

Original Code:

```
public synchronized void aMethod()
{ ... }
```

RSK Mutant:

```
public void aMethod()
{ ... }
```

RSB - Remove Synchronized Block

Similar to the RSK operator, the RSB operator removes the synchronized keyword from around a statement block which can cause a no lock bug. For example:

Original Code:

```
    synchronized(this){
     <statement_c1>
    }
```

RSB Mutant:

```
    //synchronized(this) is removed
    <statement_c1>
     ...
```

RVK - Remove Volatile Keyword

The volatile keyword is used with a shared variable and prevents operations on the variable from being reordered in memory with other operations. In the below example we remove the volatile keyword from a shared long variable. If a long variable, which is 64-bit, is not declared volatile then reads and writes will be treated as two 32-bit operations instead of one operation. Therefore, the RVK operator can cause a situation where a nonatomic operation is assumed to be atomic. For example:

Original Code:

```
volatile long x;
```

RVK Mutant:

```
long x;
```

RFU - Remove Finally Around Unlock

The finally keyword is important in releasing explicit locks. In the below example, finally ensures that the unlock() method call will occur after a try block regardless of whether or not an exception is thrown. If finally is removed the unlock() will not occur in the presence of an exception and cause a blocking critical section bug.

Original Code:

```
private Lock lock1
   = new ReentrantLock();
...
lock1.lock();
try{
   ...
} finally{
   lock1.unlock();
}
...
```

RFU Mutant:

```
private Lock lock1
   = new ReentrantLock();
...
lock1.lock();
try{
   ...
}
lock1.unlock();
...
```

5.4.4 Switch Concurrent Objects

When multiple instances of the same concurrent class type exist we can replace one concurrent object with the other.

RXO - Replace One Concurrency Mechanism-X with Another

When two instances of the same concurrency mechanism exist we replace a call to one with a call to the other. For example, consider the replacement of Lock method calls:

Original Code:

```
private Lock lock1
   = new ReentrantLock();
private Lock lock2
   = new ReentrantLock();
...
lock1.lock();
...
```

RXO Mutant for Locks:

```
private Lock lock1
   = new ReentrantLock();
private Lock lock2
   = new ReentrantLock();
...
//should be call to lock1.lock()
lock2.lock();
...
```

We can also apply the RXO operator when 2 or more objects exist of type Semaphore, CountDownLatch, CyclicBarrier, Exchanger, and more. For example consider the application of the RXO operator with two Semaphores and two Barriers:

Original Code:

```
private final Semaphore sem1
   = new Semaphore(100, true);
private final Semaphore sem2
   = new Semaphore(50, true);
...
sem1.acquire();
...
```

RXO Mutant for Semaphores:

```
private final Semaphore sem1
   = new Semaphore(100, true);
private final Semaphore sem2
   = new Semaphore(50, true);
...
//should be call to sem1.acquire()
sem2.acquire();
...
```

Original Code:

```
final CyclicBarrier bar1
   = new CyclicBarrier(20,
            new Runnable() {...});
final CyclicBarrier bar2
   = new CyclicBarrier(20,
            new Runnable() {...});
...
bar1.await();
...
```

RXO Mutant for Barriers:

```
final CyclicBarrier bar1
    = new CyclicBarrier(20,
            new Runnable() {...});
final CyclicBarrier bar2
    = new CyclicBarrier(20,
            new Runnable() {...});
...
//should be call to bar1.await()
bar2.await();
...
```

EELO - Exchange Explicit Lock Object

We have already seen the exchanging of two implicit lock objects in a synchronized block and the potential for deadlock (Section 5.4.1). The EELO operator is identical only it exchanges two explicit lock object instances:

Original Code:

```
private Lock lock1
   = new ReentrantLock();
private Lock lock2
   = new ReentrantLock();
...
lock1.lock();
...
lock2.lock();
...
finally{
   lock2.unlock();
}
...
finally{
   lock1.unlock();
}
...
```

EELO Mutant:

```
private Lock lock1
   = new ReentrantLock();
private Lock lock2
   = new ReentrantLock();
...
lock2.lock();
...
lock1.lock();
...
finally{
   lock2.unlock();
}
...
finally{
   lock1.unlock();
}
...
```

5.4.5 Modify Critical Region: Shift, Expand, Shrink and Split

The modify critical region operators cause the modification of the critical region by moving statements both inside and outside the region and by dividing the region into multiple regions.

SHCR - Shift Critical Region

Shifting a critical region up or down can potentially cause interference bugs by no longer synchronizing access to a shared variable. An example of shifting a synchronized block up is provided below. The SHCR operator can also be applied to shift up or down critical regions using other concurrency mechanisms.

Original Code:

```
<statement n1>
<statement n2>
synchronized (this){
    //critical region
    <statement c1>
    <statement c2>
}
<statement n3>
<statement n4>
...
```

SHCR Mutant:

```
<statement n1>
<statement n2>
//critical region
<statement c1>
synchronized (this){
    <statement c2>
    <statement n3>
}
<statement n4>
...
```

The SHCR operator can also be used to shift the critical region created by an explicit lock. For example:

Original Code:

```
private Lock lock1
   = new ReentrantLock();
...
public void m1 (){
<statement n1>
<statement n2>
lock1.lock();
try{
    //critical region
    <statement c1>
    <statement c2>
} finally{
    lock1.unlock();
}
<statement n3>
...
```

SHCR Mutant:

```
private Lock lock1
   = new ReentrantLock();
...
public void m1 (){
<statement n1>
lock1.lock();
try{
    <statement n2>
    //critical region
    <statement c1>
} finally{
    lock1.unlock();
}
<statement c2>
<statement n3>
...
```

EXCR - Expand Critical Region

Expanding a critical region to include statements above and below the statements required to be in the critical region can cause performance issues by unnecessarily reducing the degree of concurrency. For example:

Original Code:

```
<statement n1>
<statement n2>
synchronized (this){
    //critical region
    <statement c1>
    <statement c2>
}
<statement n3>
<statement n4>
...
```

EXCR Mutant:

```
<statement n1>
synchronized (this){
    <statement n2>
    //critical region
    <statement c1>
    <statement c2>
    <statement n3>
}
<statement n4>
...
```

The EXCR operator can also cause correctness issues and consequences such as deadlock when an expanded critical region overlaps with or subsumes another critical region.

SKCR - Shrink Critical Region

Shrinking a critical region will have similar consequences (interference) to shifting a region since both the SHCR and SKCR operators move statements that require synchronization outside the critical section. Below we provide an example of the SKCR operator using a Lock.

Original Code:

```
private Lock lock1
   = new ReentrantLock();
...
public void m1 (){
<statement n1>
lock1.lock();
try{
    //critical region
    <statement c1>
    <statement c2>
    <statement c3>
} finally{
    lock1.unlock();
}
<statement n2>
...
```

SKCR Mutant:

```
private Lock lock1
   = new ReentrantLock();
...
public void m1 (){
<statement n1>
//critical region
<statement c1>
lock1.lock();
try{
    <statement c2>
} finally{
    lock1.unlock();
}
<statement c3>
<statement n2>
...
```

SPCR - Split Critical Region

Unlike the SHCR or SKCR operators, splitting a critical region into two regions will not cause statements to move outside of the critical region. However, the consequences of splitting a critical region into 2 regions is potentially just as serious since a split may cause a set of statements that were meant to be atomic to be nonatomic. For example, in between the two split critical regions another thread might be able to acquire the lock for the region and modify the value of shared variables before the second half of the old critical region is executed.

Original Code:

```
<statement n1>
synchronized (this){
    //critical region
    <statement c1>
    <statement c2>
}
<statement n2>
...
```

SPCR Mutant:

```
<statement n1>
synchronized (this){
    //critical region
    <statement c1>
}
synchronized (this){
    <statement c2>
}
<statement n2>
...
```

5.4.6 Bug Pattern Classification of *ConMAn* Operators

In the above subsections we have provided an overview of concurrency mutation operators for Java (J2SE 5.0). In our discussion of each operator we have briefly mentioned the bug pattern that relates to that operator. Table 5.5 provides a summary of this relationship and shows that the operators we propose are examples of real bug patterns. Overall almost all of the bug patterns are covered by the operators demonstrating that the proposed concurrency operators are not only representative but provide good coverage. The bug patterns that do not have mutation operators are typically more specific complex patterns and the development of general operators related to these patterns is not feasible.

Concurrency Bug Pattern	Mutation Operators
Nonatomic operations assumed to be atomic bug pattern	RVK, EAN
Two-stage access bug pattern	SPCR
Wrong lock or no lock bug pattern	MSP, ESP, EELO, SHCR, SKCR, EXCR, RSB, RSK, ASTK, RSTK, RCXC, RXO
Double-checked locking bug pattern	–
The sleep() bug pattern	MXT, RJS, RTXC
Losing a notify bug pattern	RTXC, RCXC
Notify instead of notify all bug pattern	RNA
Other missing or nonexistent signals bug pattern	MXC, MBR, RCXC
A “blocking” critical section bug pattern	RFU, RCXC
The orphaned thread bug pattern	–
The interference bug pattern	MXT, RTXC, RCXC
The deadlock (deadly embrace) bug pattern	ESP, EXCR, EELO, RXO, ASK
Starvation bug pattern	MSF, ELPA
Resource exhaustion bug pattern	MXC
Incorrect count initialization bug pattern	MXC

Table 5.5: Concurrency bug patterns vs. *ConMAn* mutation operators

5.5 Summary

We have presented our set of *ConMAn* mutation operators to be used in the comparison of different test suites and testing strategies for concurrent Java as well as different fault detection techniques for concurrency. Although we are primarily interested in our *ConMAn* operators as comparative metrics we believe that these operators can also serve a role similar to method and class level mutation operators as both comparative metrics and coverage criteria. Our new concurrency operators should be viewed as a complement not a replacement for the existing operators used in tools like MuJava. For example, using the *ConMAn* operators can cause direct concurrency bugs while using the method and class level operators can cause indirect concurrency bugs. We define direct concurrency bugs as bugs that result from mistakes in the source code that manage concurrency mechanisms like synchronization and access to shared variables. We define indirect concurrency bugs as bugs that occur elsewhere in the source code that may have an effect on concurrency mechanisms elsewhere in a program.

We believe that our *ConMAn* operators are comprehensive and representative of real bugs. We have justified the operators by comparing them to a set of bug patterns that have been used to identify real bugs in concurrent Java programs. Additionally, our classification of the *ConMAn* operators shows that the operators are well distributed across the majority of bug patterns.

Chapter 6

A Quantitative Comparison of ConTest versus Java PathFinder

The goal of this chapter is to demonstrate our methodology by comparing two existing fault detection techniques – namely concurrency testing with the IBM tool ConTest [EFN$^+$02] and model checking with NASA's Java PathFinder (JPF) [HP00, VHB$^+$03, JPF]. With respect to these two tools we ask the following questions:

Under which circumstances is one technique more effective than another?

Under which circumstances is one technique more efficient than another?

Recall that *effectiveness* refers to the ability of each tool to detect concurrency faults and *efficiency* refers to how quickly each tool is capable of finding faults. Our interest in exploring the relationship between testing and model checking tools is motivated by a need for improved quality assurance techniques for concurrent industrial code. Testing has been an effective industrial technique for fault detection of sequential programs while model checking tools offer the potential to substantially aid in the debugging of concurrent programs.

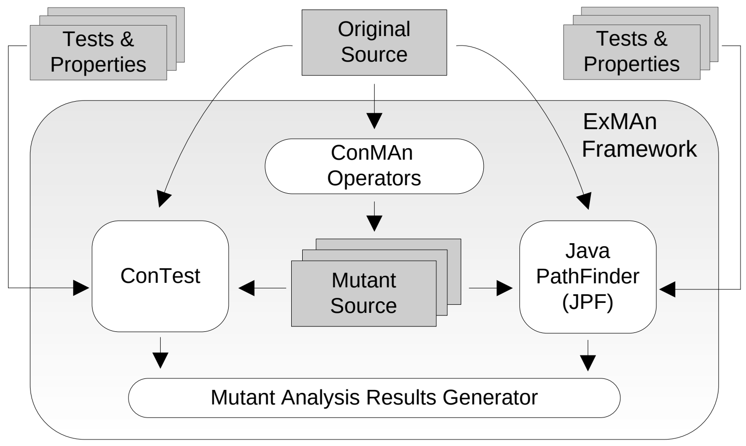


Figure 6.1: Experimental mutation analysis using the *ExMAn* framework and the *ConMAn* operators

We conducted a controlled experiment to compare ConTest and Java PathFinder. Our research approach for comparing testing and model checking is the methodology presented in Chapter 3. We use the *ExMAn* framework from Chapter 4 in combination with the *ConMAn* operators from Chapter 5 to automate the research methods used in our controlled experiment (see Figure 6.1).

In Section 6.1 we define our experimental goals. We outline our experimental setup in Section 6.2 and our experimental procedure in Section 6.3. Our experimental results and outcomes are given in Section 6.4 and threats to validity are discussed in Section 6.5. Finally, we review related work and present our conclusions in Sections 6.6 and 6.7.

6.1 Experimental Definition

The goal of our controlled experiment is to statistically evaluate ConTest and Java PathFinder using measurements to determine both the effectiveness the efficiency of each tool at finding faults.

In terms of effectiveness, there are two outcomes that are most likely. One, ConTest and Java PathFinder are *complementary*. For example, it may be the case that Java PathFinder can find bugs that ConTest cannot find and vice versa. Two, ConTest and Java PathFinder are *alternatives*. For example, it may be the case that both tools are equally likely to find most of the faults in a concurrent program. In this situation the use of both techniques in combination would provide very little, if any, benefit over either approach in isolation.

In terms of efficiency, there are three possible outcomes that are most likely. One, it might be the case that overall ConTest or Java PathFinder is *more efficient*. Two, a *mixed result* is possible where ConTest is more efficient in certain cases and Java PathFinder is more efficient in other cases. Three, it may be the case that there is *no statistical difference* between the efficiency of ConTest and Java PathFinder.

In order to determine the actual outcome of our controlled experiment we collected 3 measurements: mutant score, ability to kill and cost to kill (see Section 3.2.1). We also conducted a χ^2 test to compare the distribution of mutants detected and not detected in both ConTest and Java PathFinder. The results of this test give us insight into the effectiveness of the different techniques. Specifically, if the techniques have the same distribution they are most likely alternatives and if they have different distributions they are most likely complementary. We also conduct two-tailed and one-tailed paired t-tests to compare the time required to find a mutant in ConTest and Java PathFinder. A two-tailed test will enable us to assess if the techniques are different with respect to efficiency. If this is the case, a one-tailed test will enable us to conclude if one of the techniques is more efficient than the other.

6.2 Experimental Setup

In the section we define the setup of our experiment based on our methodology's experimental setup guidelines outlined in Section 3.3. The setup involves selecting the approaches under comparison, the example programs used in the experiment, the mutation operators used to generate faults, the quality artifacts used by the approaches under comparison and

Approach	**Stateless or stateful?**	**Exploration technique**	**Avoid searching equivalent paths?**
ConTest	stateless	random via timing delays	no
JPF Depth-First Search	stateful	systematic and exhaustive	yes
JPF Random Simulation	stateless	random via interpretive approach	no

Table 6.1: Approaches under comparison: ConTest, JPF Depth-First Search, JPF Random Simulation

the experimental environment. As we describe the selection of each part of the experimental setup we will justify our choices by answering the questions regarding each that are outlined in our methodology.

6.2.1 Selection of Approaches for Comparison

We have provided a general outline of concurrency testing with ConTest in Section 2.2.1 and model checking with Java PathFinder in Section 2.2.2. We will now briefly review the important features of each tool as they pertain to our experiment (see Table 6.1).

ConTest. A concurrency testing tool that allows for random exploration of program interleavings by inserting random timing delays in the Java bytecode. The random timing delays perturb the scheduler and each execution of a ConTest instrumented program will produce a random interleaving. ConTest is not a stateful technique and the lack of stored previous states means that there is no guarantee in ConTest that equivalent paths will not be explored. ConTest does however store the timing of the random delays for a given interleaving in order to allow play-back.

Java PathFinder (JPF) with Depth-First Search. Using Java PathFinder with a depth-first search will systematically and exhaustively explore the entire state space of the abstracted program. JPF Depth-First Search stores states (stateful) and use partial order reduction to avoid equivalent paths. We use the default scheduler in Java PathFinder since we are interested in comparing ConTest and Java PathFinder using their default settings.

Java PathFinder (JPF) with Random Simulation. We have chosen to include a second version of Java PathFinder in our experiment. Recall that Java PathFinder can be customized with different search algorithms and schedulers which explore the program state space in different ways. Using Java PathFinder with a random search and a random scheduler will explore only one random path through the state space. The random path explored is comparable to a random execution in ConTest. The difference between JPF Random Simulation and ConTest is that ConTest perturbs the scheduler while JPF Random Simulation directly controls the scheduler via an interpretive approach.

Are the approaches or tools intended for the same type of applications? Both ConTest and Java PathFinder are intended to be used with concurrent Java applications indicating that they are appropriate tools for comparison.

Do the approaches or tools have similar goals? Both ConTest and Java PathFinder can be used for the detection of faults in concurrent applications. However, one difference in terms of the goals of each tool is that ConTest is not intended to automatically detect deadlock faults while Java PathFinder is intended to find deadlocks in addition to other types of faults. To allow our testing approach to find deadlock we combine ConTest with the built-in Java Virtual Machine's (JVM) Ctrl-Break handler. The Ctrl-Break handler provides a thread dump of a running program and performs deadlock analysis on the program reporting any deadlocks detected. Figure 6.2 outlines the combined use of ConTest and the Ctrl-Break handler in the context of our experimental procedure.

6.2.2 Selection of Example Programs

We selected 4 example programs from the IBM Concurrency Benchmark [EU04] for our experiment:

- *BufWriter:* A simulation program that contains a number of threads that write to a buffer and one thread that reads from the buffer.

- *LinkedList:* A program that has two threads adding elements to a shared linked list.
- *TicketsOrderSim:* A simulation program in which agents sell airline tickets.
- *AccountProgram:* A banking simulation program where threads are responsible for managing accounts.

Table 6.2 provides metrics regarding the size of each example program.

Are the example programs representative of the kinds of programs each approach is intended for? Selecting our example programs from an existing benchmark was our best opportunity to find examples that are representative of concurrent Java applications. However, it is important to note that the programs in the benchmark use `synchronized` blocks and methods to protect access to shared data (see Table 6.3) and none of the programs use the new J2SE 5.0 concurrency mechanisms. We had difficulty finding example programs that used these mechanisms and plan to conduct further experiments in the future once these mechanisms become more widely used. Another reason for not including the new concurrency mechanisms is that they are not fully supported in ConTest. Therefore, our example programs are not representative of all concurrent Java programs written with J2SE 5.0 mechanisms like semaphores, built-in thread pools and atomic variables.

Are the example programs developed by an independent source? The example programs in the IBM Benchmark were all developed by independent sources. However, the IBM Concurrency Benchmark is maintained by some of the same researchers who developed ConTest. Although this may be perceived as a bias we believe that the fact that the actual developers of the example programs were not ConTest developers mitigates this possibility.

In order to facilitate the example programs use in our experiment, source code modifications were required because all of the programs in the benchmark had existing faults and our mutation-based setup requires *correct* original programs to compare with mutants. Therefore we modified each of the programs by hand to fix existing faults. In fixing the

Java Program:	**loc**	**classes**	**methods**	**statements**
BufWriter	213	5	9	55
BufWriter.java	62	1	1	16
Buffer.java	26	1	1	4
Checker.java	42	1	3	9
SyncWriter.java	40	1	2	13
OtherSyncWriter.java	43	1	2	13
LinkedList	303	5	22	70
BugTester.java	64	1	1	20
MyLinkedListItr.java	30	1	4	5
MyLinkedList.java	121	1	11	31
MyListBuilder.java	64	1	4	10
MyListNode.java	24	1	2	4
TicketsOrderSim	75	2	3	21
Main.java	19	1	1	5
Bug.java	56	1	2	16
AccountProgram	145	3	7	40
Main.java	59	1	1	17
Account.java	49	1	5	12
ManageAccount.java	40	1	1	11

Table 6.2: General size metrics for the example programs

Java Program:	**synch blocks**	**synch block stmts**	**synch mthds**	**synch mthd stmts**	**critical regions**	**critical region stmts**
BufWriter	3	20 (36.4%)	0 (0%)	0 (0%)	3	20 (36.4%)
BufWriter.java	0	0 (0%)	0 (0%)	0 (0%)	0	0 (0%)
Buffer.java	0	0 (0%)	0 (0%)	0 (0%)	0	0 (0%)
Checker.java	1	4 (44.4%)	0 (0%)	0 (0%)	1	4 (44.4%)
SyncWriter.java	1	8 (61.5%)	0 (0%)	0 (0%)	1	8 (61.5%)
OtherSyncWriter.java	1	8 (61.5%)	0 (0%)	0 (0%)	1	8 (61.5%)
LinkedList	2	4 (5.7%)	0 (0%)	0 (0%)	2	4 (5.7%)
BugTester.java	0	0 (0%)	0 (0%)	0 (0%)	0	0 (0%)
MyLinkedListItr.java	0	0 (0%)	0 (0%)	0 (0%)	0	0 (0%)
MyLinkedList.java	1	2 (6.5%)	0 (0%)	0 (0%)	1	2 (6.5%)
MyListBuilder.java	0	0 (0%)	0 (0%)	0 (0%)	0	0 (0%)
MyListNode.java	1	2 (50%)	0 (0%)	0 (0%)	1	2 (50%)
TicketsOrderSim	1	6 (28.6%)	0 (0%)	0 (0%)	1	6 (28.6%)
Main.java	0	0 (0%)	0 (0%)	0 (0%)	0	0 (0%)
Bug.java	1	6 (37.5%)	0 (0%)	0 (0%)	1	6 (37.5%)
AccountProgram	2	3 (7.5%)	3 (42.9%)	5 (12.5%)	5	8 (20%)
Main.java	0	0 (0%)	0 (0%)	0 (0%)	0	0 (0%)
Account.java	2	3 (25%)	3 (60%)	5 (41.7%)	5	8 (66.7%)
ManageAccount.java	0	0 (0%)	0 (0%)	0 (0%)	0	0 (0%)

Table 6.3: Concurrency metrics for the example programs

faults we were careful to use the same synchronization techniques used in other parts of the program.

6.2.3 Selection of Mutation Operators

In our experiment we decided to use a subset of the *ConMAn* operators (see Chapter 5) that mutate the Java concurrency mechanisms prior to J2SE 5.0.

Are the mutation operators systematically created from an existing fault classification? Recall that the *ConMAn* operators were created based on an existing fault classification – the concurrency bug pattern taxonomy [FNU03].

Are the mutants generated by the mutant operators the same type of faults detectable using the approaches? The operators are also ideal because they generate mutants that ConTest and Java PathFinder are capable of detecting. Our only concern regarding the capability of ConTest and Java PathFinder to detect the mutant faults was the ability of ConTest to detect mutant faults that may cause deadlock. We have alleviated this concern by using ConTest with the JVM's Ctrl-Break handler as described in Section 6.2.1.

6.2.4 Selection of Quality Artifacts

In our experiment we use test inputs and exceptions (properties) for both ConTest and Java PathFinder. Table 6.4 shows the breakdown of test inputs used for each program. For all of the programs the number of threads is also one of the input values. To ensure that the programs are representative of real programs with respect to thread count we divide the programs in Table 6.4 with respect to three thread count categories [DPE06]: small (< 5), medium ($[5, 10]$), high (> 10). Table 6.5 shows the exceptions used in each of our example programs.

Are the artifacts of any approach more mature or advanced? Do the artifacts of one approach provide an advantage over the artifacts of another approach? In order to ensure that testing with ConTest or model checking with Java PathFinder does not use more mature

Thread Count	Program with Inputs
< 5	LinkedList < # builder threads = 2, max array size= 10>
[5, 10]	TicketsOrderSim <# agent threads = 5, ticket buffer =2>
> 10	AccountProgram <# accounts = 10> BufWriter <# reader thread = 1, # writer threads = 11>

Table 6.4: Example programs categorized by thread count

Program	Properties
TicketOrderSim	(Num_Of_Seats_Sold > Maximum_Capacity)
LinkedList	this.size() != this._maxsize
BufWriter	checker.getWrittenCount() != buf._count
AccountProgram	ManageAccount.accounts[k].amount != 380

Table 6.5: Example program properties

We have already seen the use of the TicketOrderSim property in Figure 2.1. All of the other properties are checked in their respective programs just prior to termination.

quality artifacts we have decided to use the same artifacts for both techniques. Specifically, we use fixed test inputs and built-in exceptions when testing and model checking. On the one hand, testing with ConTest can detect properties that are throw at run-time. On the other hand, Java PathFinder can search the state space of a program with respect to specific test input.

6.2.5 Selection of Experimental Environment

The environment used for the experiment was a single user, single processor machine (Pentium 4 3GHz) with 3 GB of memory running the ubuntu Linux operating system.

Are there any factors in the experimental environment that can give one approach an advantage? Are there any other factors that could affect the results of the experiment in general? We chose a system with a single processor to eliminate an unfair environmental factor. The version of Java PathFinder used in our experiments does not include a multi-threaded state space search while testing with ConTest can take advantage of multiple processors. Therefore, conducting the experiment in a multi-processor environment would provide ConTest with an unfair advantage in terms of efficiency. In the future we would like to compare ConTest with a version of Java PathFinder that uses a parallel randomized state-space search [DEPP07].

We chose a single user machine because we will use real time instead of CPU time as the primary measure of efficiency. We use real time because ConTest utilizes random delays (e.g., sleep()) which are not captured when measuring only CPU time. In a multi-user environment we can not control the effect of other user processes on the analysis time of ConTest and Java PathFinder.

6.3 Experimental Procedure

After the experimental setup there are three main steps required to complete our empirical study: mutant generation, analysis (analyzing the original program and the mutants with each technique) and the collection and display of assessment results. We will now discuss the mutant generation and analysis steps. The collection and display of results will be discussed in Section 6.4 when we provide the experimental outcome.

6.3.1 Mutant Generation

The first step in our procedure is to use the *ConMAn* operators to generate mutants for each of the 4 example programs. Table 6.6 provides details on the number of each type of mutant generated for all of our example programs. The TicketsOrderSim program had the least number of mutants (3) and the BufWriter program had the most number of mutants (18). For some of the mutant operators no mutants were generated. For example, the Remove Notify All (RNA) operators generated zero mutants indicating that there were no notifyAll() calls in any of the example programs. Table 6.7 shows the distribution of mutants in each class of each example program.

ConMAn Op	**Tickets-OrderSim**	**Linked-List**	**Buf-Writer**	**Account-Program**
MXT	0	0	0	0
MSP	0	0	3	1
ESP	-	-	-	-
MSF	-	-	-	-
MXC	-	-	-	-
MBR	-	-	-	-
RTXC	0	0	2	1
RCXC	0	0	0	1
RNA	0	0	0	0
RJS	0	0	1	0
ELPA	-	-	-	-
EAN	-	-	-	-
ASTK	0	0	0	0
RSTK	0	0	0	0
ASK	0	0	3	1
RSK	0	0	0	3
RSB	1	2	3	2
RVK	0	0	0	0
RFU	-	-	-	-
RXO	-	-	-	-
EELO	-	-	-	-
SHCR	1	0	0	0
SKCR	0	0	3	0
EXCR	0	0	0	0
SPCR	1	2	3	1
TOTAL	**3**	**4**	**18**	**9**

Table 6.6: The number of mutants generated for each example program

For each operator the total number of mutants generated (including equivalent mutants) is given. Operators with values of "-" indicate that the operator does not produce any mutants for the programs because it mutates J2SE 5.0 mechanisms not present in our example programs

Java Program	**Number of Mutants**	**Number of Mutants Per Class**	**Percentage of Classes with Mutants**
BufWriter	18	3.6	80%
LinkedList	4	0.8	40%
TicketsOrderSim	3	1.5	50%
AccountProgram	9	3	66.7%

Table 6.7: The distribution of mutants for each example program

6.3.2 Analysis

In our experiment we conduct 3 kinds of analysis:

- **ConTest with the JVM Ctrl-Break Handler** (see Figure 6.2). To evaluate a mutant with ConTest we run the mutant once and then normalize the observable output via a sort. We then compare the mutant's normalized output with the original programs normalized output and see if there is a difference. If a difference is detected we move on to the next mutant. While we are running the mutant with ConTest we also use the JVM's Ctrl-Break handler to periodically check if a deadlock has occurred. The amount of time between checks is the amount of time required to run the original program to completion. If a deadlock is found in the output of the Ctrl-Break handler we stop the testing and proceed to the next mutant.

- **Java PathFinder Depth-First Search** (see Figure 6.3). To evaluate a mutant with Java PathFinder using a depth-first search we model check the program and determine if a property violation has occurred or a deadlock has been detected. If we identify a violated property or deadlock we move on to the next mutant.

- **Java PathFinder Random Simulation** (see Figure 6.4). To evaluate a mutant with Java PathFinder using a random simulation is very similar to testing a mutant with ConTest. The main difference is that unlike ConTest, Java PathFinder is capable of automatically detecting deadlocks. First, we execute the mutant once with Java PathFinder and normalize the output. Next we compare the output of the mutant with the output of the original program. If the output is different, if a property is violated or if a deadlock is detected then we have killed the mutant and we proceed with the next mutant.

For the analysis of each mutant with each technique we allow a maximum of 30 minutes to detect the fault. If no fault is identified after 30 minutes the analysis is terminated.

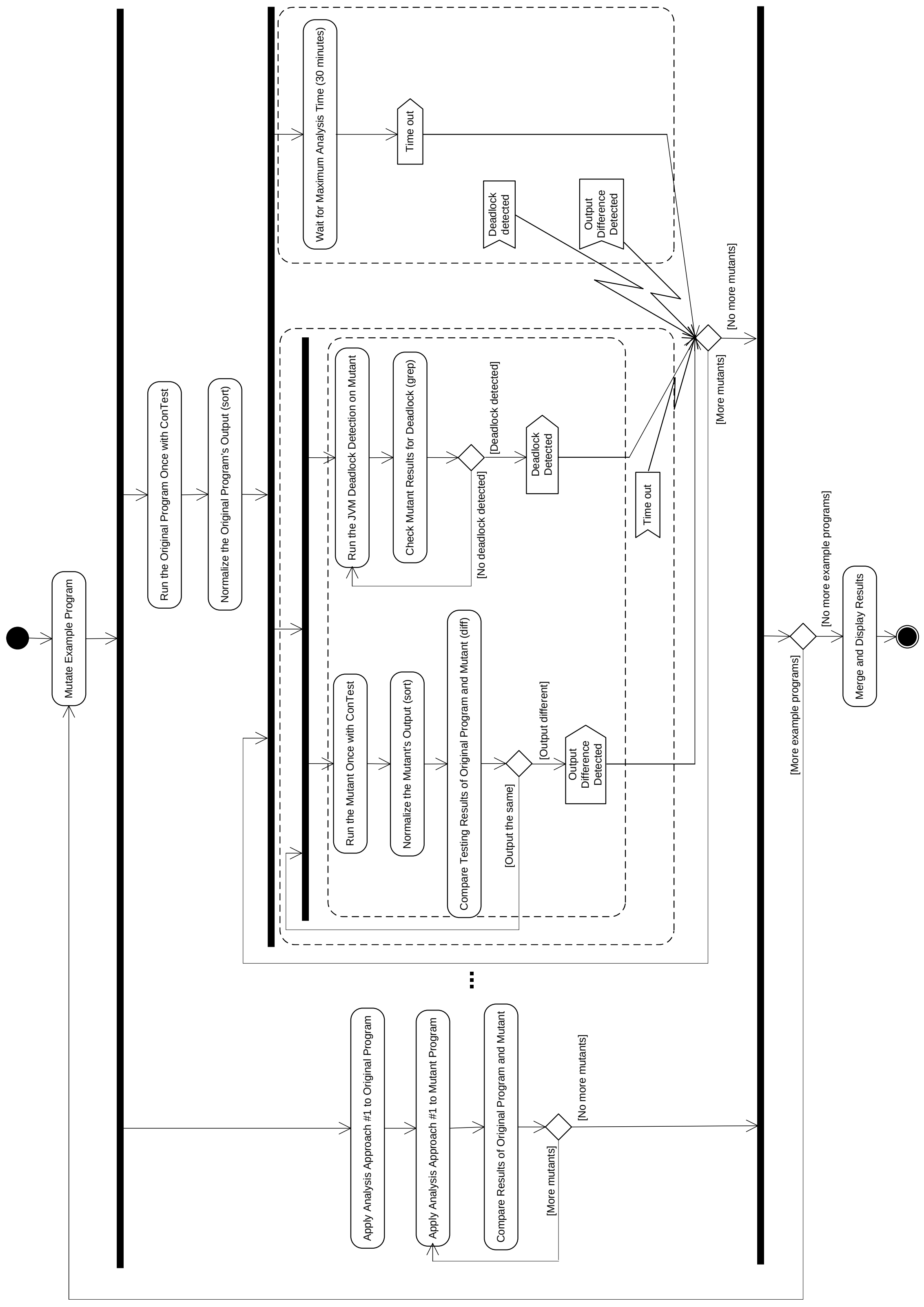

Mutate Example Program
Run the Original Program Once with ConTest
Normalize the Original Program's Output (sort)
Wait for Maximum Analysis Time (30 minutes)
Time out
Deadlock detected
Output Difference Detected
[No more mutants]
[More mutants]
Run the JVM Deadlock Detection on Mutant
Check Mutant Results for Deadlock (grep)
[Deadlock detected]
[No deadlock detected]
Deadlock Detected
Time out
Run the Mutant Once with ConTest
Normalize the Mutant's Output (sort)
Compare Testing Results of Original Program and Mutant (diff)
[Output different]
[Output the same]
Output Difference Detected
[More example programs]
[No more example programs]
Merge and Display Results
...
Apply Analysis Approach #1 to Original Program
Apply Analysis Approach #1 to Mutant Program
Compare Results of Original Program and Mutant
[No more mutants]
[More mutants]

Figure 6.2: Activity diagram of testing procedure

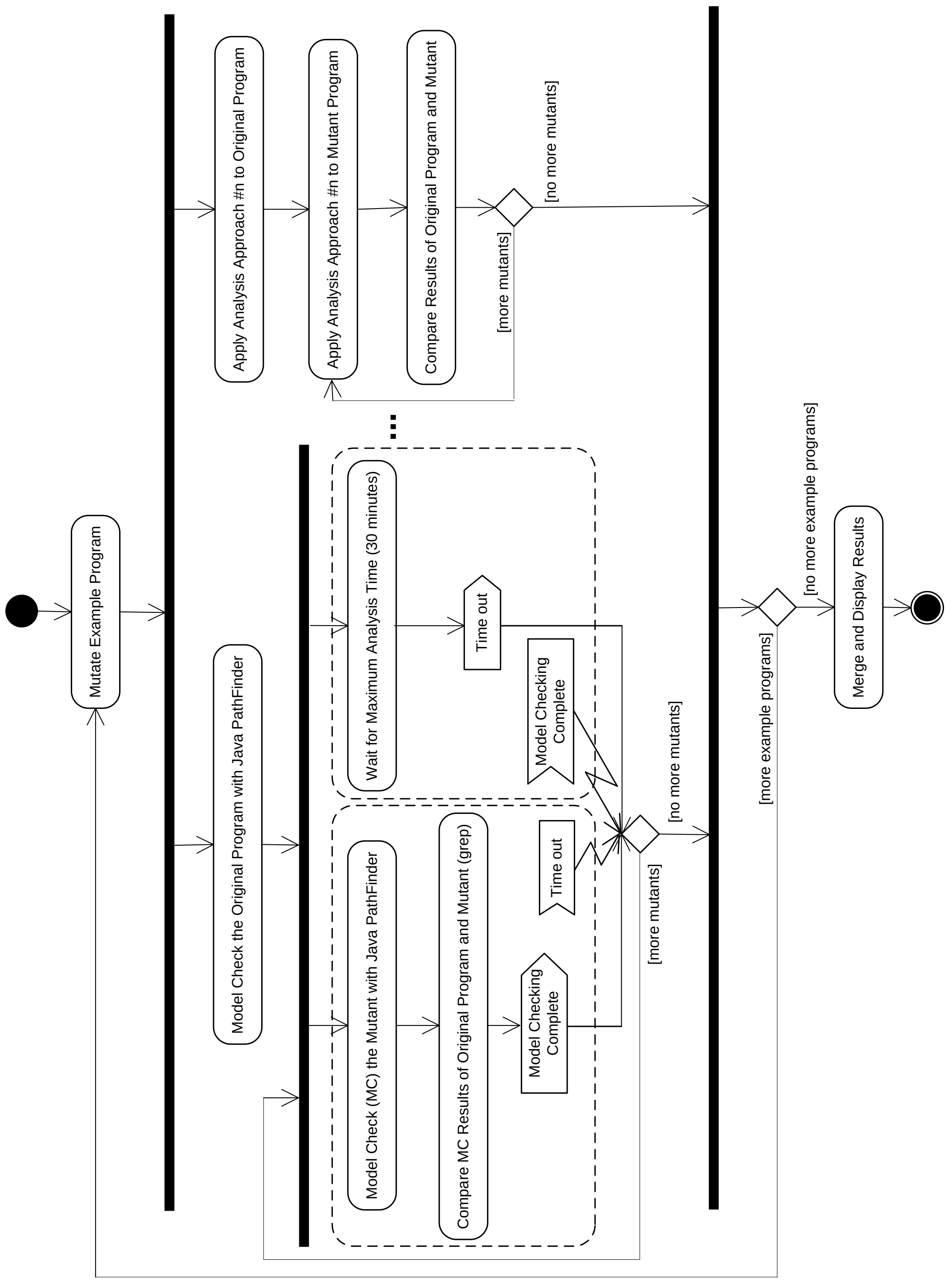

Mutate Example Program
Apply Analysis Approach #n to Original Program
Apply Analysis Approach #n to Mutant Program
Compare Results of Original Program and Mutant
[no more mutants]
[more mutants]
Model Check the Original Program with Java PathFinder
Wait for Maximum Analysis Time (30 minutes)
Time out
Model Checking Complete
Model Check (MC) the Mutant with Java PathFinder
Compare MC Results of Original Program and Mutant (grep)
Time out
Model Checking Complete
[no more mutants]
[more mutants]
[no more example programs]
[more example programs]
Merge and Display Results

Figure 6.3: Activity diagram of model checking procedure with depth-first search

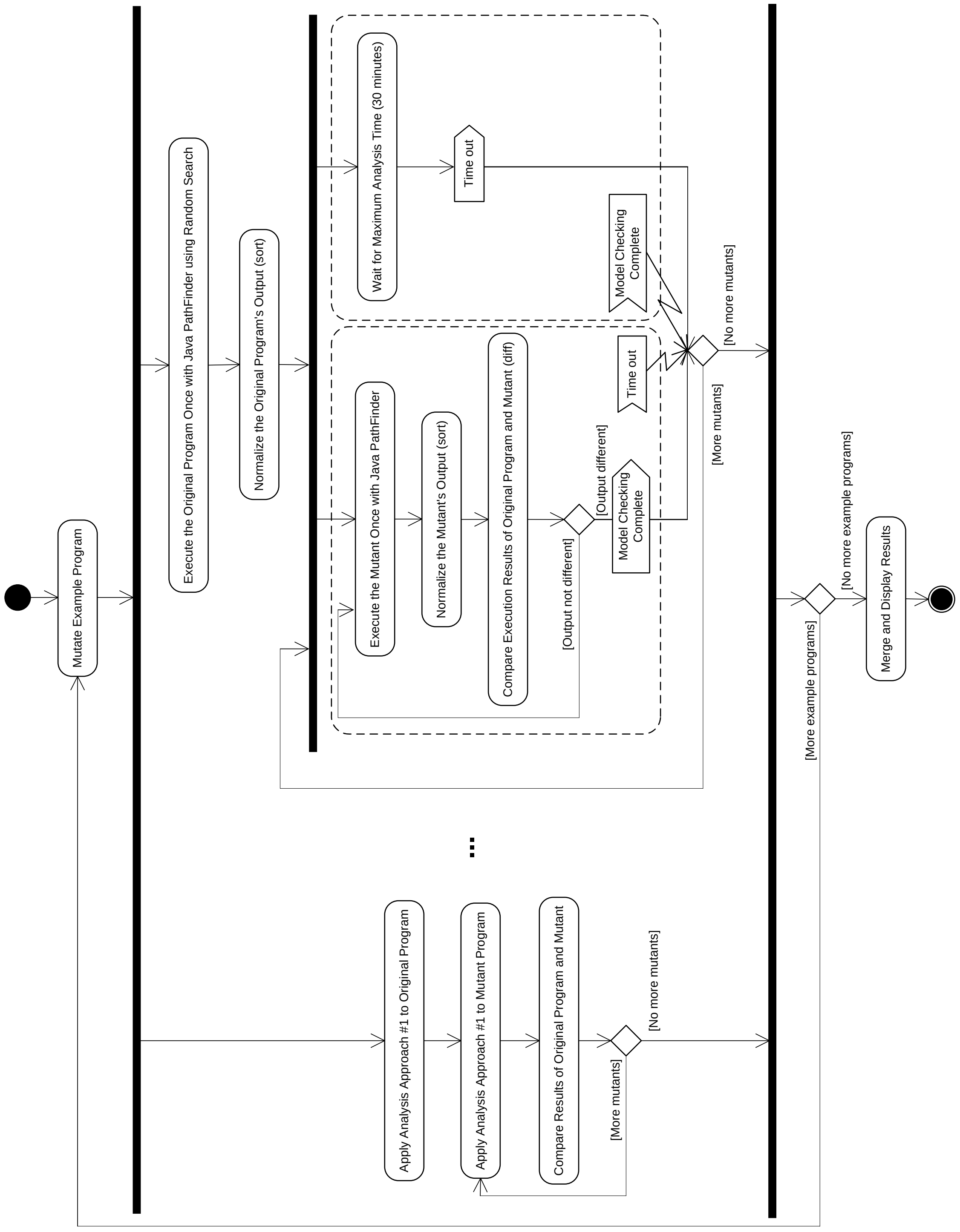

Mutate Example Program
Execute the Original Program Once with Java PathFinder using Random Search
Normalize the Original Program's Output (sort)
Wait for Maximum Analysis Time (30 minutes)
Time out
Model Checking Complete
[No more mutants]
Execute the Mutant Once with Java PathFinder
Normalize the Mutant's Output (sort)
Compare Execution Results of Original Program and Mutant (diff)
Time out
[Output different]
Model Checking Complete
[Output not different]
[More mutants]
[No more example programs]
[More example programs]
Merge and Display Results
...
Apply Analysis Approach #1 to Original Program
Apply Analysis Approach #1 to Mutant Program
Compare Results of Original Program and Mutant
[No more mutants]
[More mutants]

Figure 6.4: Activity diagram of model checking procedure with random simulation

6.4 Experimental Outcome

This section provides an overview of our experimental outcome.

6.4.1 Effectiveness

There are two possible outcomes for the effectiveness of ConTest, Java PathFinder Depth-First Search (JPFD) and Java PathFinder Random Simulation (JPFR) on our example programs. All three tools might be alternatives and capable of finding the same mutant faults or at least two of the tools might be complementary and be beneficial to use in combination. To assess the effectiveness of ConTest, JPF Depth-First Search and JPF Random Simulation at detecting faults we use the results of the mutant score and ability to kill measurements.

Using the *ConMAn* operators we generated a total of 34 mutants for our four example programs. ConTest, JPF Depth-First Search and JPF Random Simulation were all able to detect 19 mutants each (a mutant score of 56%). Tables 6.9 and 6.10 give mutant scores for different combinations of the three tools. The mutant score for each of the four example programs is given in Table 6.8. The reason the mutant scores are not higher is that some of the mutants may not be detectable using the test inputs and properties used in the experiment and some mutants may be equivalent. We have chosen to leave in these mutants because the identification of equivalent mutants is undecidable and estimating if a mutant is equivalent for concurrent programs is very difficult. Although the inclusion of these mutants may distort the effectiveness of a given technique we believe it does not affect the use of the mutation operators as a comparative metric. In Figure 6.5 we provide a more detailed view of how mutant faults were detected by ConTest, JPF Depth-First Search and JPF Random Simulation. Most of the faults in the three tools were detected by property violations. However one interesting thing to note is that ConTest and JPF

		ConTest		JPF(Depth)		JPF(Random)	
Example Program	**No. of Mutants**	**Mutants Killed**	**Mutant Score**	**Mutants Killed**	**Mutant Score**	**Mutants Killed**	**Mutant Score**
BufWriter	18	7	39%	9	50%	7	39%
LinkedList	4	2	50%	2	50%	2	50%
TicketsOrderSim	3	3	100%	3	100%	3	100%
AccountProgram	9	7	78%	5	56%	7	78%
TOTAL	34	19	56%	19	56%	19	56%

Table 6.8: The mutant scores of ConTest, JPF Depth-First Search and JPF Random Simulation for each example program

		ConTest+JPF(Depth)		ConTest+JPF(Rand.)		JPF(Depth+Rand.)	
Example Program	**No. of Mutants**	**Mutants Killed**	**Mutant Score**	**Mutants Killed**	**Mutant Score**	**Mutants Killed**	**Mutant Score**
BufWriter	18	9	50%	9	50%	10	55.6%
LinkedList	4	2	50%	2	50%	2	50%
TicketsOrderSim	3	3	100%	3	100%	3	100%
AccountProgram	9	7	78%	7	78%	7	78%
TOTAL	34	21	62%	21	62%	22	65%

Table 6.9: The mutant scores of ConTest + JPF Depth-First Search, ConTest + JPF Random Simulation and JPF Depth-First Search + JPF Random Simulation for each example program

		ConTest+JPF(Depth)+JPF(Random)	
Example Program	**No. of Mutants**	**Mutants Killed**	**Mutant Score**
BufWriter	18	10	56%
LinkedList	4	2	50%
TicketsOrderSim	3	3	100%
AccountProgram	9	7	78%
TOTAL	34	22	65%

Table 6.10: The mutant scores of ConTest in combination with JPF Depth-First Search and JPF Random Simulation for each example program

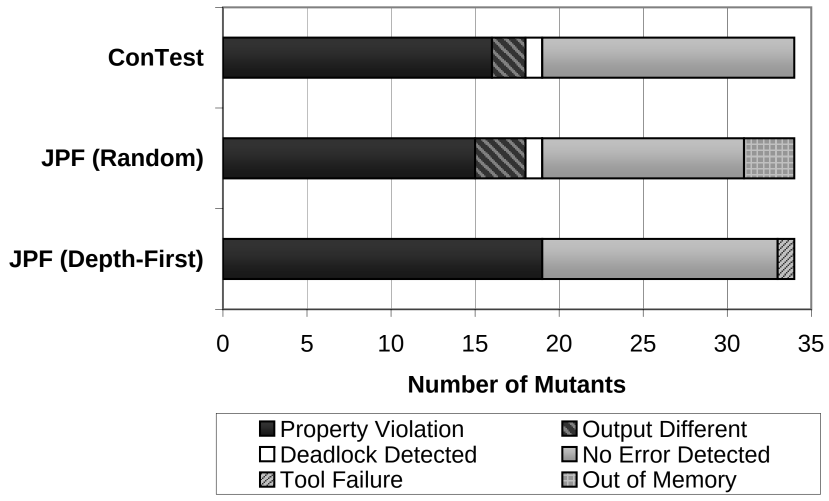


Figure 6.5: Detailed mutant results for ConTest, Java PathFinder Depth-First Search and Java PathFinder Random Simulation

Random Simulation were able to detect a deadlock that was not detected by JPF Depth-First Search due to time out. The reason there were not more deadlocks detected by the tools is that the example programs were all small in size and typically did not contain nested critical regions. Mutation of nested critical regions is most likely to produce mutants that will cause deadlock.

To determine if the distribution of mutants killed and not killed by ConTest, JPF Depth-First Search and JPF Random Simulation is the same we used a χ^2 test. The null hypothesis for the test was: measurements from ConTest, JPF Depth-First Search, JPF Random Simulation are from the same distribution. The test produced a low χ^2 value (0.00) indicating that at the standard confidence level of 0.05 we can not reject the null hypothesis. Although the percentage of mutant faults detected by all three tools is identical we have still not determined if ConTest, JPF Depth-First Search and JPF Random Simulation are alternative or complementary for our example programs.

To assess if ConTest, JPF Depth-First Search and JPF Random Simulation are alternatives or complementary we need to consider the ability of each tool to kill (detect) different

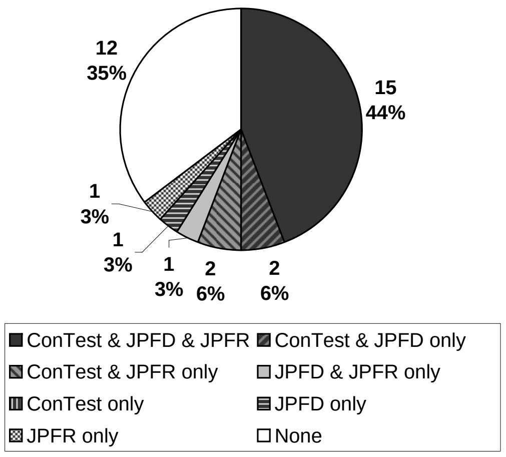


Figure 6.6: ConTest, JPF Depth-First Search and JPF Random Simulation: mutants detected by all three tools, two tools, one tool or neither tool

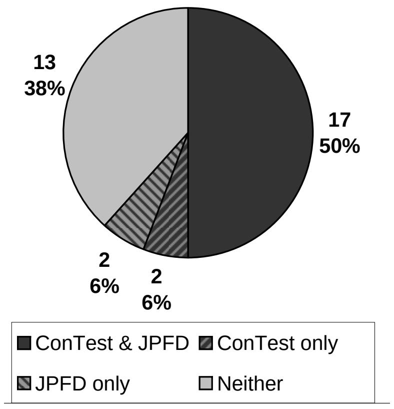


Figure 6.7: ConTest and JPF Depth-First Search: mutants detected by both, one or neither tool

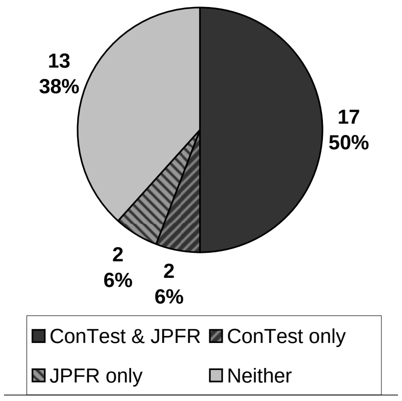


Figure 6.8: ConTest and JPF Random Simulation: mutants detected by both, one or neither tool

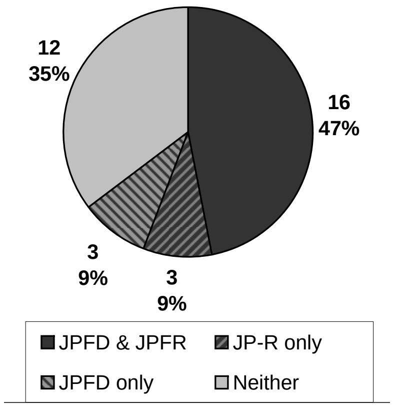


Figure 6.9: JPF Depth-First Search and JPF Random Simulation: mutants detected by both, one or neither tool

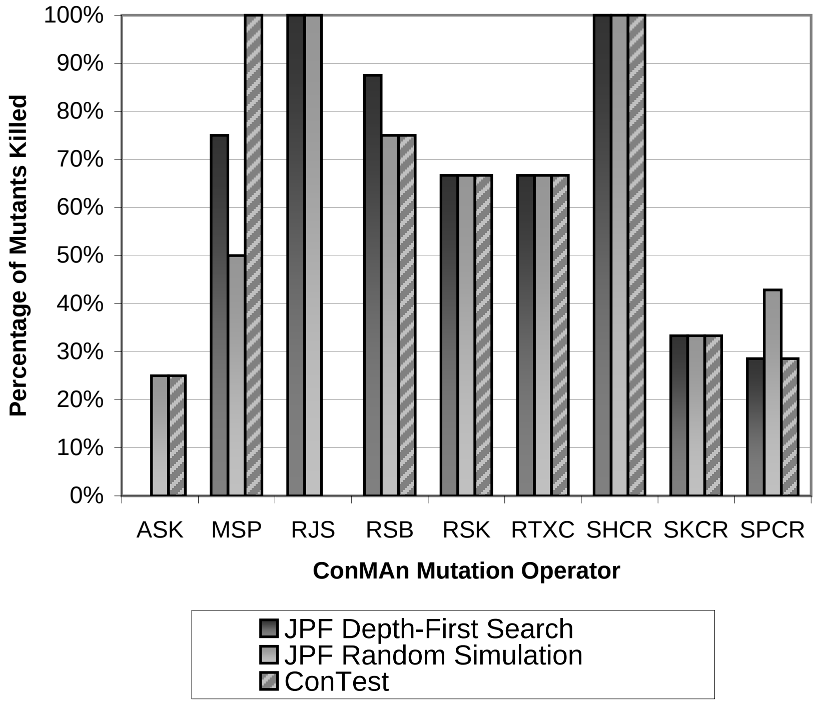


Figure 6.10: Ability to detect (kill) each kind of mutant in the example programs

kinds of mutants. Figure 6.10 is a bar graph which shows the percentage of mutants generated by each *ConMAn* operator that are killed by ConTest, JPF Depth-First Search and JPF Random Simulation. The graph shows that there are some variations in the ability of ConTest and the two configurations of Java PathFinder at the mutant operator level. For example, ConTest and JPF Random Simulation found a higher percentage of Add Synchronized Keyword to Method (ASK) mutants. ConTest also found a higher percentage of Modify Synchronized Block Parameter (MSP) mutants than JPF Depth-First Search and JPF Random Simulation. JPF Depth-First Search and JPF Random Simulation were more successful then ConTest with respect to the Replace Join() with Sleep() (RJS) mutant operator, JPF Depth-First Search found a higher percentage of Remove Synchronized Block (RSB) mutants than ConTest and JPF Random Simulation and finally JPF Random Simulation found a higher percentage of Split Critical Region (SPCR) mutants. For all other types of mutants the three tools found the same percentage. Figure 6.6 provides an analysis of the number of mutants detected by all three tools, two of the tools, one tool or neither tool. Figures 6.7, 6.8 and 6.9 provide comparison of each distinct pair of tools. Of the 34 mutants, 15 (44%) were detected by all three tools, 5 (15%) were detected by two of the tools, 1 (3%) was detected by JPF Depth-First Search only and 1 (3%) was detected by JPF Random Simulation only. All of the mutants detected by ConTest were detected by JPF Depth-First Search or JPF Random Simulation.

The mutant score of using ConTest, JPF Depth-First Search and JPF Random Simulation in combination is 65%, 9% higher than using any of the tools in isolation. Since all of the mutants detected by ConTest were detected by JPF Depth-First Search or JPF Random Simulation the combined mutant score of the two configurations of Java PathFinder was also 65%. The improved mutant score overall seems to indicate that the tools are complementary and their combined usage is beneficial for the example programs. However, when we consider the mutant scores for each example program the two tools seem to be alternatives. Table 6.8 shows the mutant scores for each program. In both the LinkedList and the TicketsOrderSim programs the mutant scores are the same for all three tools together and in isolation. For the AccountProgram, ConTest and JPF Random Simulation were able to detect 2 more mutants (ASK, MSP) than JPF Depth-First Search and the combined use of the three tools achieved the same mutant score as the best tool in isolation. The BufWriter program was the only example program that showed an improvement in the mutant score due to combined used of two or more of the tools. With the BufWriter program JPF Depth-First Search and JPF Random Simulation combined achieved a mutant score of 56%, the same as ConTest, JPF Depth-First Search and JPF Random Simulation in combination. JPF Depth-First Search achieved the best mutant score in isolation – 50%. For three of the example programs the combined use of ConTest, JPF Depth-First Search and JPF Random Simulation achieved the same mutant score as the better of the three tools in isolation and for one of the examples there was a 6% increase in the mutant score if the tools were used in combination. This result indicates that for our example programs that ConTest, JPF Depth-First Search and JPF Random Simulation are most likely *alternatives*.

6.4.2 Efficiency

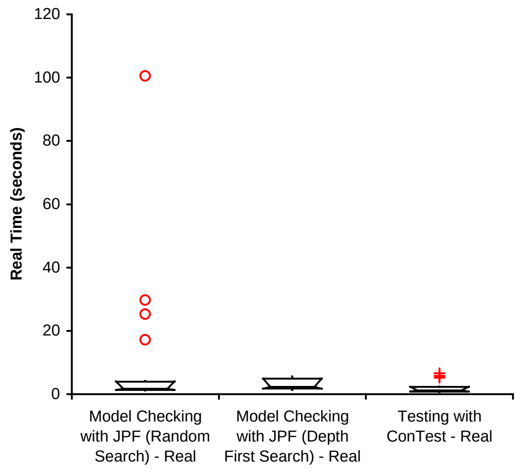
There are 3 possible outcomes with respect to the efficiency of ConTest, JPF Depth-First Search and JPF Random Simulation at detecting faults in our example programs. One of the tools might be more efficient, there may be no difference in the efficiency or there may be a mixed result. To visually compare the cost to kill a mutant we present box plots in Figure 6.11. For each box plot the middle line in the box indicates the median value and the ends of the box are the first and third quartile. Additionally, we will also individually compare each pair of tools using a paired t-test. We are interested in comparing each combination of tools because it will potentially provide more understanding of the efficiency of the tools than a comparison of all three tools together. To determine the efficiency outcome for each pair of tools we used the cost to kill a mutant. For example, for all mutant faults detected by both ConTest and JPF Depth-First Search we compared the real time to detect a mutant for each tool.

ConTest and JPF Depth-First Search. From the box plots it appears that ConTest is more efficient at detecting a fault when all three techniques are capable of finding the fault. Specifically, ConTest has a smaller median and smaller first and third quartiles.

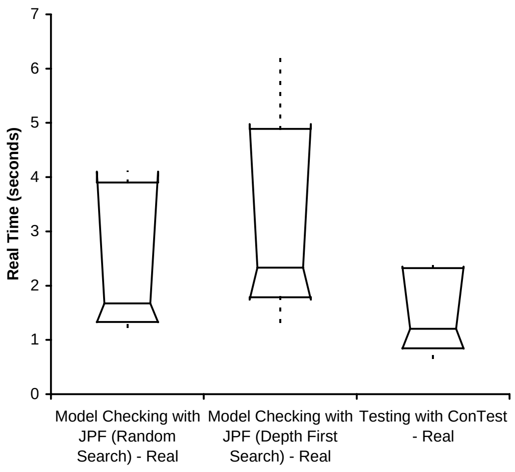
Based on the box-plot we tested the proposition that the cost to kill a mutant using JPF Depth-First Search was less than using ConTest. We tested the proposition using a 1-tailed paired t-test[1]. We were able to conclude that at the 0.05 level our proposition was not correct (p-value = 0.0085).

ConTest and JPF Random Simulation. Based on the box-plot we tested the propositions that the cost to kill a mutant using JPF Random Simulation was the same as

[1]In order to perform a paired t-test an important assumption is that the difference between the test data must by normally distributed. That is, the difference in detection times for each mutant using ConTest and JPF Depth-First Search must be normal. We used the Shapiro-Wilk test to assess normality.



(a) Box plot with outliers



(b) Box plot without outliers

Figure 6.11: Box plots of cost to detect (kill) mutants for ConTest, JPF Depth-First Search and JPF Random Simulation

the cost to kill a mutant with ConTest. We were unable to conclude that at the 0.05 level that there our proposition was incorrect (p-value = 0.1693).

JPF Depth-First Search and JPF Random Simulation. Based on the box-plot we tested the propositions that the cost to kill a mutant using JPF Depth-First Search was the same as the cost to kill a mutant with JPF Random Simulation. We were unable to conclude that at the 0.05 level that there our proposition was incorrect (p-value = 0.2432).

In summary the results of our box plot and paired t-test conclude that JPF Depth-First Search is *not more efficient* than ConTest for our example programs. For the comparison of ConTest and JPF Random Simulation and the comparison of JPF Depth-First Search and JPF Random Simulation we were unable to conclude that one technique was more efficient than the other. However, based on the raw cost to kill data it appears that in some cases one tool may be better and in others another tool may be more efficient, thus indicating a *mixed* result.

6.5 Threats to Validity

There are a number of issues of validity to be considered: internal validity, external validity, construct validity, and conclusion validity [WRH^{+}00]. We have previously discussed in Chapter 3 how our methodology tries to mitigate these threats.

Internal validity. With respect to internal validity, it is also important to address three factors that Dwyer, Person, and Elbaum identified as having a significant influence on the performance of path-sensitive error detection techniques [DPE06]:

1. Variations in search order: these variations *"...can give rise to very large variations in path-sensitive analysis cost and fault detection effectiveness across a range of programs"* [DPE06]. Our goal was to compare Java PathFinder and ConTest in their default configurations. JPF Depth-First Search does not use a random scheduler by default while JPF Random Simulation and ConTest do use random scheduling. In the future we plan to also evaluate JPF Depth-First Search with random scheduling. For this experiment we follow the advice of Dwyer, Person and Elbaum and state the scheduling used by each approach so it is clear that this factor may affect our experimental outcome.

2. Path error density in example programs: *"...programs with high path error density will exhibit almost no variation in analysis across different path-sensitive analysis techniques"* [DPE06]. Dwyer, Person and Elbaum measured path error density, the probability of an error path occurring in the thread schedule, by running a given program between 1000 and 10000 times and recording the number of executions that exhibited an error. They used the JPF Random Simulation configuration used in this study to collect their findings. We have not measured the path error density of our mutation programs and it may affect the validity of the experiment if all of the mutants generated for each example program have high error densities.

3. Number of threads in example programs: *"Programs with a high number of threads impacted cost only when density was medium or low..."* [DPE06]. In our experiment we have ensured a selection of programs with low, medium and high thread counts (see Table 6.4).

Another factor that may affect internal validity is the time bound of 30 minutes placed on the analysis using each tool. In some cases (see Figure 6.5) both ConTest and Java PathFinder were unable to find mutant faults in the 30 minutes allowed and extending the time bound may affect the analysis results.

External validity. In our experiment there are three major threats to external validity. First, a threat to external validity is possible if the mutant faults used do not adequately represent real faults for the programs under experiment. We ensure representative mutants by using the *ConMAn* operators which are based on an existing fault model [FNU03]. Second, a threat to validity is possible if the software being experimented on is not representative of the software to which we want to generalize. In our experiment the small set of programs are not representative of all concurrent Java applications and therefore our results do not generalize well. Third, an additional threat to the validity is that the configurations of Java PathFinder and ConTest used in our experiment limit our ability to generalize to each approach. For example, Java PathFinder can be customized with other search algorithms and scheduling strategies that may affect both its effectiveness and efficiency with respect to fault detection. In a recent study, Dwyer, Person, and Elbaum concluded that the search order used in a tool can influence the effectiveness of the analysis [DPE06].

Construct validity. There is a potential for threats to construct validity if ConTest and Java PathFinder are not used in the way in which they are intended. We discussed this issue briefly when we outlined the importance of selecting tools with similar goals that are applied to the same kind of applications. Ensuring this is the case limits the need to modify how the tools are used.

Conclusion validity. In order to reduce threats to conclusion validity in our experiment we need to have confidence that our measurements are correct and the statistical tests are used correctly. First, we limit threats by increasing the chances that our measurements are recorded correctly by automating the collection of measurements using our *ExMAn* framework. Second, we have done our best to satisfy the statistical test assumptions of the χ^2 and paired t-tests.

6.6 Related Work

Several empirical studies, discussed in Section 2.4, have focused on the ability of testing and model checking to find faults. For example, the Chockler et al. case study compared ConTest and the ExpliSAT model checker using two real programs at IBM [CFG$^+$06]. Overall, ConTest was found to be easier to use but was not as a comprehensive in identifying potential problems in the software. The comprehensiveness considered by Chockler et al. is a similar measurement to our effectiveness. One difference between this research and our own is that Chockler et al. do not consider the efficiency of each tool.

The Brat el al. controlled experiment compared traditional testing, runtime analysis, model checking and static analysis [BDG$^+$04]. Recall that although no statistically significant conclusions were drawn from the experiment the authors stated that the results *"...confirmed our belief that advanced tools can out-perform testing when trying to locate concurrency errors"* [BDG$^+$04]. We achieved a different outcome in our experiment that compared the same model checker (Java PathFinder) with testing. We believe the primary difference for our results is the kind of testing used. On the one hand, we used testing with ConTest, which is a sophisticated tool that automatically seeds time delays into Java byte code to explore different interleavings. On the other hand, Brat el al. used standard black box system testing. Exploration of different interleavings was not automatic – instead the testing relied on native scheduling differences by using different operating systems with different Java Virtual Machines. Furthermore, manual instrumentation of delays as well as thread priorities were used.

Our work differs from this previous work in that we are able to draw statistically significant conclusions regarding both the effectiveness and efficiency of testing (with ConTest) and model checking (with Java PathFinder) at finding mutant faults. Although the previous comparisons did not have statistically significant quantitative results it is important to acknowledge their contributions to the community and to our own work since we build upon these previous studies.

6.7 Conclusion

We have presented a controlled experiment that uses program mutation to compare the fault detection capabilities of testing with ConTest and model checking with Java PathFinder. The experimental procedure is automated using our *ExMAn* framework and the use of mutation with concurrent Java is supported by our *ConMAn* operators. The experiment demonstrates the utility of program mutation as an aid to understanding testing and model checking of concurrent software.

Our experiment has tried to better understand the effectiveness and efficiency of ConTest and Java PathFinder at detecting mutant faults in our example programs. With respect to effectiveness, we conclude that ConTest and Java PathFinder (depth-first search and random simulation) are most likely *alternative* fault detection techniques rather than complementary with respect to the programs used. However, the ability to kill measurements show that the tools do not detect all of the same kinds of mutants equally. With other example programs there maybe a potential to use ConTest and Java PathFinder in a complementary way. With respect to efficiency, we conclude that ConTest is more efficient and can detect a mutant in less time on average than JPF Depth-First Search for our example programs. We were unable to conclude if there was a difference in efficiency between

ConTest and JPF Random Simulation or if there was a difference in efficiency between Java PathFinder with a depth-first or random simulation. However, the overall times to detect mutants using all the tools were fairly small.

It is important to be clear that the results of our experiments do not generalize to all concurrent Java software. The example programs are relatively small in size and do not contain any of the new J2SE 5.0 concurrency mechanisms. However, the results still provide insight into the relationship between testing with ConTest and model checking with Java PathFinder and the usefulness of each in detecting bugs and improving the quality of concurrent software. In order to achieve stronger and more general conclusions we need to conduct further experiments. We need to compare testing with ConTest and model checking with JPF using larger sets of example programs and different configurations of each tool in order to build a strong body of empirical results.

Chapter 7

Summary and Conclusions

7.1 Summary

In Chapter 1 we introduced the goal of this thesis – to show that mutation analysis is a useful technique for the comparison of different fault detection techniques and to establish a supporting methodology and framework. A secondary goal was also introduced – to better understand the effectiveness and efficiency of testing and model checking with respect to fault detection in concurrent Java source code. In Chapter 2 we reviewed the background for this thesis including an overview of existing fault detection techniques for concurrent Java (e.g., testing with ConTest, model checking with Java PathFinder) and a survey of existing empirical software engineering research in the areas of program mutation and fault detection techniques.

In this thesis we developed a methodology for the assessment of fault detection techniques like testing and model checking using measurements to compare the effectiveness and efficiency of each technique at finding bugs (Chapter 3). Specifically, we used program mutation to determine how many bugs a technique can find and at what speed. In Chapter 4 we implemented our methodology and create the *ExMAn* framework. In Chapter 5 we created the *ConMAn* operators for concurrent Java which are used with the *ExMAn* framework to facilitate our comparison of testing with ConTest and model checking with Java PathFinder in Chapter 6.

7.2 Contributions

The contributions of this thesis include:

1. The development of a *generalized methodology* for using mutation to conduct controlled experiments of different quality assurance approaches with respect to fault detection. To the best of our knowledge our proposed approach is novel since no other work has used mutation at the source code level as a method of comparing property-based analysis techniques with testing.

2. The implementation of the *ExMAn (EXperimental Mutation ANalysis) framework* automates and supports our methodology. *ExMAn* is a reusable implementation for building different customized mutation analysis tools for comparing different quality assurance techniques. For example, in Chapter 6 we used one customized configuration of *ExMAn* to compare ConTest and Java PathFinder for detecting bugs in concurrent Java programs. The main contribution of *ExMAn* includes its abilities to act as an enabler for further research in the community.

3. The design and implementation of the *ConMAn (CONcurrency Mutation ANalysis) operators* for applying program mutation with concurrency and specifically with concurrent Java applications. When comparing testing and model checking for concurrent systems in *ExMAn* we used the *ConMAn* operators to create faulty mutant versions of example programs. The application of the *ConMAn* operators to programs from the IBM Concurrency Benchmark also provides the community with a large set of new programs to use in evaluating concurrent Java applications.

4. *Empirical results* on the effectiveness and efficiency of testing with ConTest and model checking with Java PathFinder (using both a depth-first search and random simulation) as fault detection techniques for concurrent Java applications.

7.3 Limitations

In addition to the contributions of our research it is also important to point our the limitations of our work. We will discuss the limitations of both the *ExMAn* framework and our *ConMAn* operators.

- **Limitations to the *ExMAn* framework**
  - Limited to comparing fault detection techniques and tools with command-line interfaces.
  - Limited to using example programs that are sequential or concurrent but not distributed.
  - For some fault detection techniques and tools the generation of scripts to run the mutation analysis has to be modified by hand because the use of the tool does not fit perfectly with the experimental procedure shown in Figure 3.5. For example, we had to customize the script for ConTest in our experiment in Chapter 6 in order to ensure that the instrumentation step of ConTest was only done once and not repeated for every interleaving.
- **Limitations to the *ConMAn* operators**
  - One limitation to using the *ConMAn* operators with larger programs is due to the difficulty in determining if the original example program is correct. Recall, that we need to use the output of the original example program as a baseline to compare with the output of the mutant program. The larger the program the

more difficult it is to ensure that the program is correct. In Chapter 6 we used an approximation of correctness. However, it is not clear if this approximation is sufficient for all example programs.

– Detection of equivalent mutants is undecidable [HHD99] and often it is the responsibility of the human involved in the mutation testing process to identify possibly equivalent mutants. Due to the difficulty that humans have with understanding concurrency, the identification of possibly equivalent mutants generated by the *ConMAn* operators is even more challenging than for mutation operators with sequential code.

– In order to ensure that a critical region in a concurrent program is protected, defensive programming is sometimes used which results in redundant critical regions that protect the same shared data. The *ConMAn* operators are not effective in programs with redundant critical regions and using *ConMAn* with these types of programs will most likely lead to high numbers of equivalent mutants.

– The *ConMAn* operators only directly affect the code responsible for concurrency in Java and do not consider mutations to the code which may indirectly affect concurrency. In Chapter 5 we mentioned other mutation operators for Java including the method mutation which we believe are complementary to our *ConMAn* operators. These other operators can indirectly affect concurrency and could be used in combination with our concurrency operators to avoid this limitation. One potential problem with using the method mutation operators with the *ConMAn* operators is that it could lead to huge explosions of mutants.

– We are required to use strong mutation (mutant is run till termination) with the *ConMAn* operators. An alternative to strong mutation is weak mutation, which has been successful at reducing the cost of mutation with sequential programs. Weak mutation checks the state of the program after executing the mutated line of source code [OU00]. Thus, the mutant program does not need to run completely in order to determine if the mutant was killed. With concurrent programs the state of the original and the mutant program could be different after the mutant code executes as a result of different interleavings.

7.4 Future Work

We have identified three primary areas of future work: further empirical studies, validating program mutation for concurrent software and optimization of testing and model checking based on empirical assessment.

7.4.1 Further Empirical Studies

As we mentioned in the contributions section of this thesis (Section 7.2) we view the research methodology and the *ExMAn* framework as enablers for further experiments. The empirical results presented in Chapter 6 demonstrate the feasibility of our approach and contribute to our understanding about testing and model checking concurrent software systems. However, in order to achieve stronger and more general conclusions we need to increase both the depth and breadth of our experiments. On the one hand, we need to conduct more experiments comparing testing with model checking in order to build a strong body of results. On the other hand, we need to conduct a breadth of experiments comparing fault detection techniques in other ways. For example in the future we would like to conduct experiments that study other aspects of testing and model checking:

- the comparison of different model checkers and testing tools
- the effect of varying the number of interleavings for testing
- the effect of varying the levels of bounded depth for model checking
- the effect of varying the search algorithms for model checking

- the comparison of different model checkers (e.g., Java PathFinder and Bandera/Bogor)
- the comparison of different sets of properties within the same model checker (this type of comparison is similar to comparing different test suites)
- the comparison of different types of properties (assertions vs. linear temporal logic (LTL)) within the same model checker

In addition to further experiments with model checking and testing we would also like to expand the breadth of the experiments further by comparing different static analysis techniques, comparing sequential testing with static analysis and more.

7.4.2 Validating Program Mutation for Concurrent Software

Another possible area of future work is to further empirically assess and validate the use of program mutation as a technique for seeding faults in concurrent programs. Seeding faults with mutation has two important uses: one, as an assessment metric for evaluating and comparing quality assurance techniques and, two, as a mechanism for generating and optimizing software quality artifacts (e.g., tests, assertions, or temporal logic properties).

This thesis has focused on using the *ConMAn* operators with concurrent source code to compare testing and model checking. In our future work we intend to empirically study two more fundamental questions about the role of program mutation in concurrency:

1. Is detecting bugs created by mutation similar in difficulty to detecting real concurrent faults?
2. What is a sufficient set of operators for mutation? Is there a smaller set of concurrency mutation operators that can be as beneficial as using the entire set of operators developed in Chapter 5?

Addressing these questions will provide a better understanding of the benefits of using program mutation with concurrent software and provide an improved set of mutation operators for concurrency. Addressing the first question will involve an empirical assessment of program versions with real bugs and program versions with bugs generated using concurrency mutation operators. Recall that we have already demonstrated that the *ConMAn* operators are representative of real faults. This further experiment will determine if detecting mutants created with the *ConMan* operators is similar in difficulty to detecting real faults. Addressing the second question will also involve an empirical assessment of concurrent programs with bugs generated using only the concurrency mutation operators. A possible source of programs for the above experiments is the IBM concurrency benchmark which contains programs with real faults.

Previous research has addressed both of these question with respect to non-concurrent mutant operators and sequential programs [ABL05, OLR^{+}96, NA06]. The results of this previous work do not generalize to the concurrent systems domain and therefore this proposed area of future work will be a novel contribution to the research community. Currently, there is a lack of empirical studies in the area of software engineering and this work would also provide a contribution in this regard.

7.4.3 Optimization of Testing and Model Checking Based on Empirical Assessment

In the context of testing and model checking we are interested in extending our *ExMAn* framework to include components to optimize and improve the quality artifacts used by testing tools and model checkers. We have previously proposed this extension in [BCD05, Bra06] but implementing and evaluating this extension is still future work. The optimization we propose will consist of 4 steps and will be supported by the development of two components; a test suite optimization component and a property set optimization component (see Figure 7.1):

1. *Test case generation:* If there exists a mutant that is killed by a property and no test

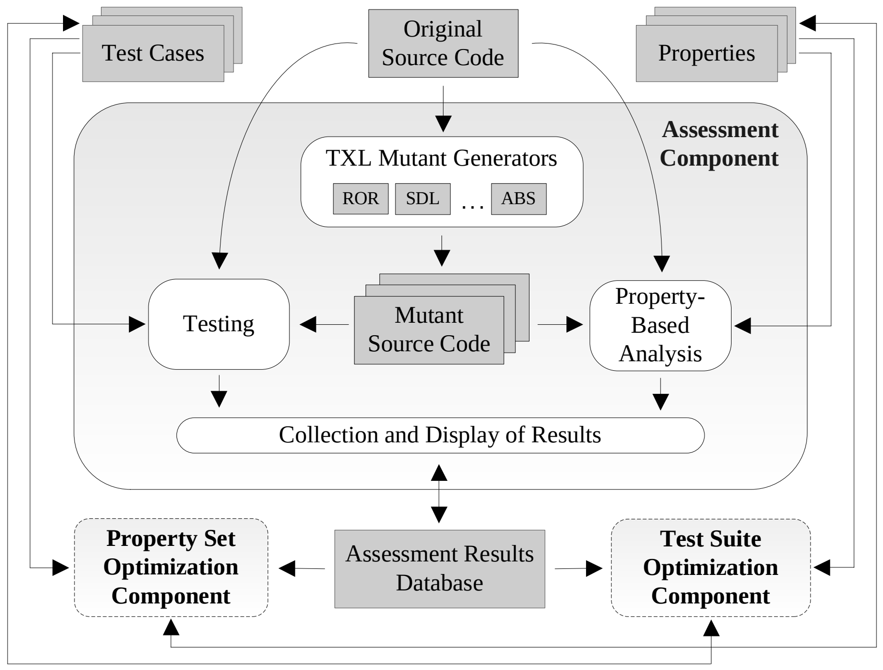


Figure 7.1: Future directions – extending the *ExMAn* framework to include optimization

case exists to kill the mutant then we plan to use a counter-example generated by the model checker to develop a test case that will kill the mutant. Existing approaches to test case generation using model checkers are described in [BCH$^+$04, HdMR04].

2. *Property generation:* If there exists a mutant that is killed by possibly several test cases and no property exists to kill the mutant the use of dynamic slicing or run-time monitoring of the test cases may assist in the generation of a property that will kill the mutant. Related work on property generation from test cases includes Daikon [ECGN01] and Terracotta [YE04].
3. *Test suite reduction:* If one property kills several mutants and several test cases kill the same mutants then we might be able to generate a test case that kills all of the mutants that can replace multiple test cases.
4. *Property set reduction:* On one hand, we will reduce the set of properties by comparing the mutants killed by each property. If the mutants killed by a given property are all killed by other properties it could be removed. On the other hand, if one test

case kills several mutants and several properties in our set kill the same mutants then we might be able to generate a single property that kills all of the mutants that can replace multiple properties.

Our approach to optimization should allow for both the test suite and property set to be enhanced for use in isolation and in hybrid approaches. Our proposed optimization should provide an improved test suite and property set that will allow the testing and property-based analysis to each be more comprehensive and succinct.

7.5 Conclusion

Many people have already argued convincingly that there is a need for more empirical, quantitative research in software engineering in general [BK99] as well as better empirical practices [JMV04, LLQ$^+$05, SDJ07]. Moreover, others have argued that there is a need not only for better empirical practices but also for more third-party experiments that do not have a self-confirmatory bias [ZMM06]. Our work has been motivated by both of these observations. In particular, we have developed an empirical methodology for comparing fault detection techniques and have implemented the methods in our *ExMAn* framework. To the best of our knowledge our proposed methodology is a novel approach since no other work has used program mutation at the source code level as a method of comparing different techniques like model checking and testing. We believe that our methodology and framework are contributions to the state-of-the-art in empirical software engineering because they provide *automatic* experimentation that is *reproducible* by third-party researchers. Furthermore, we believe that our work can provide benefits to both industry and research by evaluating existing quality assurance tools for concurrency and demonstrating how to best use them.

Our experimental results are also beneficial because, similar to the work of Dwyer, Person, and Elbaum [DPE06], we are a third-party with no bias regarding any of the tools under experimentation. There have been previous comparisons of testing and model checking [BDG$^+$04, CFG$^+$06] and our work is unique from this previous work because we are able to draw statistically significant conclusions regarding both the effectiveness and efficiency of testing (with ConTest) and model checking (with Java PathFinder) at finding mutant faults. Although the previous comparisons did not have statistically significant quantitative results it is important to acknowledge their contributions to the community and to our own work as our results build upon the results of these previous studies.

Bibliography

[ABD02] Paul Ammann, Paul E. Black, and Wei Ding. Model checkers in software testing. Technical Report NIST-IR 6777, National Institute of Standards and Technology, Feb. 2002.

[ABL05] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of* 27*th* *International Conference on Software Engineering (ICSE 2005)*, pages 402–411, May 2005.

[ABM98] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proc. of* 2*nd* *IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, 1998.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *Proc. of the* 1*st* *International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03)*, Apr. 2003.

[And04] Paul Anderson. CodeSurfer/Path Inspector. In *Proc. of the IEEE Int. Conf. on Software Maintenance (ICSM'04)*, page 508, Sept. 2004.

[AZ03] James H. Andrews and Yingjun Zhang. General test result checking with log file analysis. *IEEE Trans. Softw. Eng.*, 29(7):634–648, 2003.

[Ban] Bandera project at Kanas State University. Web page: `http://bandera.projects.cis.ksu.edu/` (last accessed: June 26, 2007).

[BCD05] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. An empirical framework for comparing effectiveness of testing and property-based formal analysis. In *Proc. of* 6th *Int. ACM SIGPLAN-SIGSOFT Work. on Program Analysis for Software Tools and Engineering (PASTE 2005)*, pages 2–5, Sept. 2005.

[BCD06a] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. ExMAn: A generic and customizable framework for experimental mutation analysis. In *Proc. of the* 2nd *Workshop on Mutation Analysis (Mutation 2006)*, pages 57–62, Nov. 2006.

[BCD06b] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Proc. of the* 2nd *Workshop on Mutation Analysis (Mutation 2006)*, pages 83–92, Nov. 2006.

[BCD07] Jeremy S. Bradbury, James R. Cordy, and Juergen Dingel. Comparative assessment of testing and model checking using program mutation. In *Proc. of the* 3rd *Workshop on Mutation Analysis (Mutation 2007)*, Sept. 2007.

[BCH$^+$04] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proc. of the* 26th *International Conference on Software Engineering (ICSE 2004)*, pages 326–335, May 2004.

[BDG$^+$04] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Păsăreanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental evaluation of verification and validation tools on Martian Rover software. *Formal Methods in Systems Design Journal*, 25(2-3):167–198, Sept. 2004.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change.* The XP Series. Addison Wesley, 2000.

[BK99] Susan S. Brilliant and John C. Knight. Empirical research in software engineering: a workshop. *SIGSOFT Softw. Eng. Notes*, 24(3):44–52, 1999.

[Bra06] Jeremy S. Bradbury. Using mutation for the assessment and optimization of tests and properties. In *Doctoral Symposium being held in conjunction with the International Symposium on Software Testing and Analysis (ISSTA 2006)*, Jul. 2006.

[CDH$^+$00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Proc. of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448. ACM Press, Jun. 2000.

[CDMS02] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the TXL transformation system. *J. of Information and Software Technology*, 44(13):827–837, 2002.

[CFG$^+$06] Hana Chockler, Eitan Farchi, Ziv Glazberg, Benny Godlin, Yarden Nir-Buchbinder, and Ishai Rabinovitz. Formal verification of concurrent software: Two case studies. In *Proc. of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD-IV)*, pages 11–21, Jul. 2006.

[Cor06] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, Aug. 2006.

[CS98] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of Java multithreaded applications. In *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '98)*, pages 48–59. ACM Press, 1998.

[DEPP07] Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. Parallel randomized state-space search. In *Proc. of the* 29th *International Conference on Software Engineering (ICSE 2007)*, pages 3–12, May 2007.

[DGK+88] R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt. An extended overview of the Mothra software testing environment. In *Proc. of the* 2nd *Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Jul. 1988.

[DHP+07] Matthew B. Dwyer, John Hatcliff, Corina Păsăreanu, Robby, and Willem Visser. Formal software analysis: Emerging trends in software model checking. In *Proc. of* 29th *International Conference on Software Engineering (ICSE 2007), 2007 Future of Software Engineering (FOSE '07)*, pages 120–136, May 2007.

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints for test data selection: help for the practicing programmer. *IEEE Computer*, 11(4):34–41, Apr. 1978.

[DM96] Márcio Eduardo Delamaro and José Carlos Maldonado. Proteum – a tool for the assessment of test adequacy for C programs. In *Proc. of the Conference on Performability in Computing Sys. (PCS 96)*, pages 79–95, Jul. 1996.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[DPE06] Matthew B. Dwyer, Suzette Person, and Sebastian Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. of the* 14th *ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*, pages 92–104. ACM Press, Nov. 2006.

[DR05] Hyunsook Do and Gregg Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proc. of the* 21st *IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 411–420, 2005.

[DR06] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. on Soft. Eng.*, 32(9):733–752, Sept. 2006.

[EA06] Steve Easterbrook and Jorge Aranda. Case studies for software engineers. Tutorial Presented at the 28th International Conference on Software Engineering (ICSE 2006). Slides available at `http://www.cs.toronto.edu/~sme/case-studies/case_study_tutorial_slides.pdf` (last accessed: June 26, 2007), May 2006.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Soft. Eng.*, 27(2):1–25, Feb. 2001.

[EFBA03] Yaniv Eytani, Eitan Farchi, and Yosi Ben-Asher. Heuristics for finding concurrent bugs. In *Proc. of the* 1st *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003)*, Apr. 2003.

[EFN+02] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[EU04] Yaniv Eytani and Shmuel Ur. Compiling a benchmark of documented multithreaded bugs. In *Proc. of the* 2nd *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, Apr. 2004.

[FNU03] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proc. of the* 1^{st} *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003)*, Apr. 2003.

[Gho02] Sudipto Ghosh. Towards measurement of testability of concurrent object-oriented programs using fault insertion: a preliminary investigation. In *Proc. of the* 2^{nd} *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 17–25, 2002.

[Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. on Soft. Eng.*, 3(4):279–290, Jul. 1977.

[HdMR04] Grégoire Hamon, Leonardo de Moura, and John Rushby. Generating efficient test sets with a model checker. In *Proc. of the Software Engineering and Formal Methods, Second International Conference on (SEFM'04)*, pages 261–270, Sept. 2004.

[HDPR02] John Hatcliff, Matthew B. Dwyer, Corina S. Păsăreanu, and Robby. Foundations of the Bandera abstraction tools. *The Essence of Computation: Complexity, Analysis, Transformation*, pages 172–203, 2002.

[HHD99] Rob Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[Hie05] Robert M. Hierons. Editorial: Validating our findings. *Software Testing, Verification and Reliability*, 15(4):209–210, Dec. 2005.

[HKHZ99] Yatin Hoskote, Timothy Kam, Pei-Hsin Ho, and Xudong Zhao. Coverage estimation for symbolic model checking. In *Proc. of the ACM/IEEE Conf. on Design Automation*, pages 300–305, 1999.

[Hoa03] C.A.R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, Apr.-Jun. 2003.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), Apr. 2000. Special issue containing selected submissions for the 4^{th} SPIN Workshop.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[IEE02] *IEEE standard glossary of software engineering terminology (610.12-1990).* IEEE, Dec. 1990 (Reaffirmed 2002).

[Jli02] *Jlint Manual: Java program checker*, Jan. 2002.

[JMV04] N. Juristo, A. M. Moreno, and S. Vegas. Towards building a solid empirical body of knowledge in testing techniques. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, 2004.

[JOW06] Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified software: a grand challenge. *IEEE Computer*, 39(4):93–95, Apr. 2006.

[JPF] Java PathFinder website. Web page: `http://javapathfinder.sourceforge.net/` (last accessed June 26, 2007).

[KM99] Heinz K. Klein and Michael D. Myers. A set of principles for conducting and evaluating interpretive field studies in information systems. *MIS Quarterly*, 23(1):67–93, 1999.

[KO91] K. N. King and A. Jefferson Offutt. A Fortran language system for mutation-based software testing. *Softw. Pract. Exper.*, 21(7):685–718, 1991.

[LDG$^+$04] Brad Long, Roger Duke, Doug Goldson, Paul A. Strooper, and Luke Wildman. Mutation-based exploration of a method for verifying concurrent Java components. In *Proc. of the* 2nd *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, Apr. 2004.

[Lea00] Doug Lea. *Concurrent Programming in Java^{TM}, Second Edition.* Addison Wesley, 2000.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33– 42, May 2006.

[LHS01] Brad Long, Dan Hoffman, and Paul Strooper. A concurrency test tool for Java monitors. In *Proc. of the* 16th *International Conference on Automated Software Engineering (ASE 2001)*, pages 421–425. IEEE Computer Society, 2001.

[LHS03] Brad Long, Daniel Hoffman, and Paul Strooper. Tool support for testing concurrent Java components. *IEEE Trans. on Soft. Eng.*, 29(6):555–566, Jun. 2003.

[LLQ$^+$05] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for evaluating bug detection tools. In *Proc. of the Workshop on the Evaluation of Software Defect Detection Tools*, Proc. available at `http://www.cs.umd.edu/~pugh/BugWorkshop05/` (last accessed June 26, 2007), June 2005.

[LSS05] Timothy C. Lethbridge, Susan Elliott Sim, and Janice Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, 10(3):311–341, 2005.

[LSW07] Brad Long, Paul Strooper, and Luke Wildman. A method for verifying concurrent Java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 19(3):281–294, Mar. 2007.

[MKO02] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *Proc. of the* 13^{th} *International Symposium on Software Reliability Engineering*, pages 352–363. IEEE Computer Society Press, Nov. 2002.

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, Jun. 2005.

[NA06] Akbar S. Namin and James H. Andrews. Finding sufficient mutation operators via variable reduction. In *Proc. of* 2^{nd} *Workshop on Mutation Analysis (Mutation 2006)*, Nov. 2006.

[OAL06] Jeff Offutt, Paul Ammann, and Lisa (Ling) Liu. Mutation testing implements grammar-based testing. In *Proc. of the* 2^{nd} *Workshop on Mutation Analysis (Mutation 2006)*, Nov. 2006.

[OAW$^+$01] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, , and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *In Proc. of* 12^{th} *International Symposium on Software Reliability Engineering (ISSRE 2001)*, pages 84–93, Nov. 2001.

[Off92] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, 1992.

[OLR$^+$96] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.

[OMK04] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. An experimental mutation system for Java. In *Proc. of the Workshop on Empirical Research in Software*

Testing (WERST'2004). Also published in the SIGSOFT Software Engineering Notes, 29(5):1–4, ACM Press, 2004.

[OU00] Jeff Offutt and Roland H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, Oct. 2000.

[Pfl95] Shari Lawrence Pfleeger. Experimental design and analysis in software engineering. *Annals of Software Engineering*, 1(1):219–253, Dec. 1995.

[PSE06] Dewayne E. Perry, Susan Elliott Sim, and Steve Easterbrook. Case studies for software engineers. In *Proc. of the* 28^{th} *International Conference on Software Engineering (ICSE 2006)*, pages 1045–1046, May 2006.

[RAF04] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *Proc. of the* 15^{th} *International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256. IEEE Computer Society, 2004.

[RB02] J.J. Ritsko and M.L. Bates. Preface. *Issue on Software Testing and Verification, IBM Systems Journal*, 42(1):2–3, 2002.

[RDH03] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. of the* 9^{th} *European Software Engineering Conference held jointly with the* 11^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11)*, pages 267–276. ACM Press, 2003.

[Roy70] Winston W. Royce. Managing the development of large software systems: Concepts and techniques. In *Technical Papers of Western Electronic Show and Convention (WesCon)*, Aug. 1970.

[RTI02] RTI. The economic impacts of inadequate infrastructure for software testing. planning report 02-3. Technical report, National Institute of Standards and Technology, Program Office Strategic Planning and Economic Group, United States, May 2002.

[Rus00] John Rushby. Disappearing formal methods. In *Proc. of the High-Assurance Systems Eng. Symp. (HASE'00)*, pages 95–96, Nov. 2000.

[SDJ07] Dag I. K. Sjøberg, Tore Dybå, and Magne Jørgensen. The future of empirical methods in software engineering research. In *Proc. of* 29th *International Conference on Software Engineering (ICSE 2007), 2007 Future of Software Engineering (FOSE '07)*, pages 358–378, May 2007.

[SEH03] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proc. of the* 25th *International Conference on Software Engineering (ICSE 2003)*, pages 74–83. IEEE Computer Society, May 2003.

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), Mar. 2005.

[TKD04] Oded Tal, Scott Knight, and Thomas R. Dean. Syntax-based vulnerability testing of frame-based network protocols. In *Proc. of the* 2nd *Annual Conference on Privacy, Security and Trust (PST 2004)*, pages 155–160, Oct. 2004.

[VHB^{+}03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, Apr. 2003.

[Woo06] Jim Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, Oct. 2006.

[WRH+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction.* Software Engineering. Kluwer Academic Publishers, 2000.

[WS06] Margaret A. Wojcicki and Paul Strooper. A state-of-practice questionnaire on verification and validation for concurrent programs. In *Proc. of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD-IV)*, pages 1–10, Jul. 2006.

[YE04] Jinlin Yang and David Evans. Dynamically inferring temporal properties. In *Proc. of the International Workshop on Program Analysis for Software Tools and Engineering (PASTE 2004)*, pages 23–28, Jun. 2004.

[ZMM06] Carmen Zannier, Grigori Melnik, and Frank Maurer. On the success of empirical studies in the International Conference on Software Engineering. In *Proc. of the* 28th *International Conference on Software Engineering (ICSE 2006)*, pages 341–350, Jun. 2006.