University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Computer Science and Engineering: Theses, Dissertations, and Student Research | Computer Science and Engineering, Department of

Summer 7-31-2014

A Methodology and Tool for Concurrent Fault Injection

ZhongYin Zhang
University of Nebraska-Lincoln, zzy4032@gmail.com

Follow this and additional works at: http://digitalcommons.unl.edu/computerscidiss

Part of the Computer Engineering Commons

Zhang, ZhongYin, "A Methodology and Tool for Concurrent Fault Injection" (2014). *Computer Science and Engineering: Theses, Dissertations, and Student Research*. 81.
http://digitalcommons.unl.edu/computerscidiss/81

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Computer Science and Engineering: Theses, Dissertations, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

A METHODOLOGY AND TOOL FOR CONCURRENT FAULT INJECTION

by

ZhongYin Zhang

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Gregg Rothermel and Witty Srisa-an

Lincoln, Nebraska

August, 2014

A METHODOLOGY AND TOOL FOR CONCURRENT FAULT INJECTION

ZhongYin Zhang, MS

University of Nebraska, 2014

Adviser: Gregg Rothermel and Witty Srisa-an

As the speed of microprocessors tails off, utilizing multiple processing cores per chip is becoming a common way for developers to achieve higher performance. However, writing concurrent programs can be a big challenge because of common concurrency faults. Because concurrency faults are hard to detect and reproduce, traditional testing techniques are not suitable. New techniques are needed, and these must be assessed. A typical method for assessing testing techniques is to embed faults in programs using mutation tools, and assess the ability of techniques to detect these. Although mutation testing techniques can be used to represent common faults, approaches for representing concurrency faults have not been created. In this paper, we introduce a methodology for injecting mutations related to concurrency faults, focusing on four common concurrency fault patterns as mutant operators. We implement the approach in the Eclipse IDE. We empirically study our approach's effectiveness by using it to seed various types of concurrency faults based on the four fault patterns in a set of programs. This approach generates many times more mutants than can be seeded by hand. We then execute the original programs and these mutants. We characterize the mutants in terms of detectability as part of our study. The results show that using the proposed tool, concurrent fault injection tool (CFIT) is feasible and efficient.

ACKNOWLEDGMENTS

I would like to thank my two advisors. Dr. Gregg Rothermel and Dr.Witty Srisa-an for their invaluable guidance, support and encouragement over the past few years.

Dr. Gregg Rothermel led me into a wonderful research area and taught me how to do rigorous research. As a good mentor, he provided me with valuable advice all the time and gave me advice and support when I ran into trouble in my life. I would like to thank him for everything he did for me.

Dr. Witty Srisa-an has also had a great influence on me. He led me into the gorgeous system's area and taught me how to be a good programmer. I would like to thank him for his infinite encouragement and patience. Without him, I can not imagine how I could achieve the goal that I previously thought was not possible. All his advising and mentoring are valuable to me and I will remember it now and forever. I do not know how I can possibly thank him enough. I hope I can repay him by making him proud of me in the future.

I would like to thank Dr. Anita Sarma for offering time to serve as my committee member, reviewing my thesis and delivering me valuable feedback and suggestion.

I would like to thank all my friends in the Esquared lab and UNL, Tingting Yu, Pingyu Zhang, Jianguo Wang, Jian Hu, Yin Guo, Miao Zhen, Nic Colgrove, Thammasak Thianniwet, Yalan Liang,etc. I can not imagine how I could survive without you through all these years. Especially, I would like to thank Jianguo Wang, Jian Hu, Yin Guo and Pingyu Zhang for huge help when I had trouble. I appreciate everything they did for me.

Finally, I would like to thank my family, Mom, Dad, and my loving spouse - Zhen Hu, for their infinite love and encouragement.

Contents

Contents **iv**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

2 Background **4**

2.1 Types of Concurrency Faults 4

2.2 Mutation Testing 7

2.3 Mutation Testing Tool 8

2.4 Eclipse 8

2.4.1 Architecture of Eclipse Plug-ins 9

2.4.1.1 Extension Points 9

2.4.1.2 Extensions 9

2.4.2 C/C++ Development Tooling (CDT) 11

2.4.2.1 Visitor Pattern API for ASTs 12

3 Design and Implementation **13**

3.1 Fault Patterns 13

3.1.1 Remove Unlock 15
3.1.2 Remove Lock 16
3.1.3 Remove Paired Lock and Unlock (Critical Section Violation) . 16
3.1.4 Switch Lock Order 17
3.2 Implementation of a Concurrency Fault Injection Tool 18
3.2.1 CFIT Architecture 18
3.2.1.1 Injection Action Extension 18
3.2.1.2 Mutation System 19
3.2.1.3 Mutant Property 20
3.2.1.4 Database 22
3.2.1.5 Hibernate 22
3.2.1.6 CFIT Working Process 23

4 Empirical Study **27**
4.1 Purpose of Study 27
4.2 Objects of Study 28
4.3 Study Operation 30
4.4 Result 30
4.5 Discussion 35

5 Conclusion and Future Work **38**

Bibliography **39**

List of Figures

2.1 Deadlock circular wait 7
2.2 Eclipse plug-ins 10
2.3 Extensions and extension points 11

3.1 Bug patterns 14
3.2 Concurrent fault types 15
3.3 Class ASTVisitor in DOM AST 21
3.4 Class LockManagementVisitor 21
3.5 CFIT procedure 23
3.6 Snapshot of programs after modifications made by CFIT 26

4.1 Experiment procedure 31

List of Tables

3.1 Concurrency Fault Taxonomy 14

4.1 Mutant Data 31

4.2 Remove Unlock 32

4.3 Remove Lock 32

4.4 Remove Paired Lock and Unlock 33

4.5 Switch Lock Order 33

4.6 The total numbers of detected mutants based on all 4 mutant operators and the number of detected mutants based on the percentage of test cases 34

4.7 Deadlocks for Base Program 35

Chapter 1

Introduction

As the speed of microprocessors tails off, utilizing multiple processing cores per chip is becoming a common way for developers to achieve higher performance. To do this, developers shift from writing sequential code to employing thread-level parallelism. Writing dependable concurrent programs can, however, be challenging, because improper synchronization of access to shared resources can lead to runtime errors such as deadlocks, critical section violations, livelock, and starvation which are difficult to detect, isolate, and correct during pre-deployment.

Typically, a concurrent program consists of two or more processes or threads that cooperate in performing a task[8]. Since there are multiple processes or threads executing simultaneously, shared variables or resources may be accessed concurrently. Without proper protection, these accesses can result in intermittent runtime errors that occur only when under specific execution interleavings or occurrences of specific events.

Currently, there are many techniques used to detect concurrency faults, such as data race detection[13][28][33], atomicity violation detection[14], pattern analysis[25], and fault-localization[26][37][31]. Moreover, common testing techniques involving per-

formance testing and stress testing are always used to deal with concurrency faults. However, performance testing and stress testing are very time consuming and it can be difficult to reproduce the concurrency faults they detect. Thus, we need better testing techniques to address concurrency issues.

Currently, researchers use mutation testing approaches to represent common but hard to detect faults, in order to make testing more efficient. Mutation testing is a fault-based software testing technique that uses mutants that slightly modify a piece of code in a program to check the quality of a new testing technique and reproduce faults that are hard to detect[16][11]. There are several existing approaches for defining mutation operators for concurrent programs[35][27][15][24][38][22]; however, these approaches still rely on using manually injected mutants and output-based test oracles.

Injecting mutants manually is neither efficient nor complete, especially when it is applied to modern concurrent software systems that tend to have large code bases. In addition, output based oracles are not sufficient because occurrences of concurrency faults do not always lead to erroneous outputs; therefore, they often elude traditional testing approaches that rely on output-based oracles for fault detection. As such, internal test oracles, which detect faults by monitoring aspects of internal program and system states[39] can be more effective for detecting these types of faults.

In previous work[39], Yuetal. empirically investigated the use of internal test oracles based on manually seeding mutants in 5 applications. The results show that internal oracles can be more effective than output-based oracles. However, due to the fact that manual seeding of mutants is time consuming and inaccurate, an automatic concurrency fault seeding tool is necessary. In this paper, we introduce an automatic concurrent fault injection tool (CFIT) based on an Eclipse plug-in for C/C++. We use four common concurrent fault patterns as mutant operators. We then empirically

study our tool's effectiveness by using it to seed various types of concurrency faults based on four fault patterns in the same five programs. This approach generates many times more mutants. We then execute the original programs and these mutants. We characterize these mutants as part of our study. The results show that using the proposed tool, CFIT, is feasible and efficient.

The remainder of this thesis is organized as follows. In Section 2, we provide background information relevant to the remainder of the thesis. We describe the design and implementation of our concurrent fault injection tool (CFIT) in Section 3. Section 4 presents our empirical study. Conclusions and future work are discussed in Section 5.

Chapter 2

Background

In this chapter, we discuss background information related to this work. First, we describe and provide examples of common concurrency faults. We then describe mutation testing approaches and existing tools to support such testing. Last, we provide an overview of the Eclipse plug-in architecture.

2.1 Types of Concurrency Faults

In this section, we describe four types of concurrency faults: critical section violations, deadlock, livelock, and starvation.

Critical section violations occur when two or more processes or threads attempt to access and update a shared resource at the same time. This situation is very common in multi-threaded or multi-process systems. This type of fault occurs when shared resources are not properly protected by lock operations that synchronize concurrent access to those resources. As an example, suppose there are two processes P1 and P2, both that can perform write operations on a variable **a**. Initially, **a** is set to 0. If **a** is not properly protected, both P1 and P2 can concurrently write to **a**. Thus, the two

processes race to update the shared resource. As such, the final value of **a** depends on who has the last access. Code snippet A provides an example of this type of common data race in an application. Function autoIncrement updates global variable **a**. In a scenario where two threads execute autoIncrement simultaneously, the final value of **a** may not be 2.

```
1.  int a = 0;
2.  void autoIncrement(){
3.  //lock();
4.     a++;
5.  //unlock();
6.  }
7.  main(){
8.     autoincrement();
    }
```

Code snippet A

Deadlock is a situation in which more than one thread or process are blocked permanently because each is waiting to access a shared resource that is blocked by one of the others at that time. There are four conditions that must be met for deadlock to occur: mutual exclusion (only one process or thread can access a shared resource in a critical section at a time), hold-and-wait (a process or thread may hold a shared resource while awaiting assignment of other resources), no preemption (no resource can be released from a process or thread holding it) and circular wait (each process or thread holds at least one shared resource requested by the other processes or threads)[36]. An example is provided in Figure 2.1. There are two shared resources, RS1 and RS2, and two processes, P1 and P2; RS1 is held by P1 and RS2 is held by P2. There is no preemption and each process has exclusive access to the held resource.

Because P2 needs RS1 which is held by P1, and P1 needs RS2 which is held by P2, a circular wait occurs. Code snippet B indicates a common instance of such a deadlock scenario in an application. If two threads are used in this program, there is a specific interleaving sequence T1(1), T2(6), T1(2), T2(7) that can cause a deadlock to occur.

```
void RS1(){
  ...
1. lock1();
2. lock2();
3. //critical section.
4. unlock2();
5. unlock1();
  ...
}
```

```
void RS2(){
  ...
6. lock2();
7. lock1();
  //critical section
8. unlock2();
9. unlock1();
  ...
}
```

Code snippet B

A livelock is similar to a deadlock except that processes or threads are not blocked permanently by each other. Rather, they are constantly processed by the CPU. An example is when a spinlock instead of blocking is used to synchronize a region. Two threads can be spinning on a lock. They are both executing on the processor but

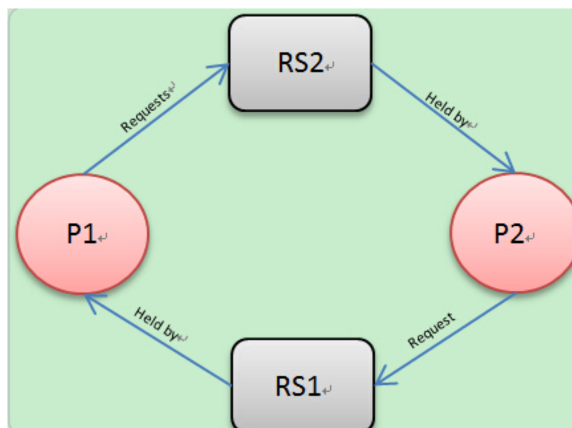

Figure 2.1: Deadlock circular wait

without making any progress toward completion.

Starvation is a situation in which a process or thread can never access shared resources. As an example, suppose three processes with three different priority levels need to access a shared resource. If the process with the highest priority keeps using the resource, the other two lower priority processes would not be able to access the resource.

2.2 Mutation Testing

Mutation testing is a fault-based software testing technique that uses mutants that slightly modify a piece of code of the program to check the quality of a new testing technique and reproduce faults that are hard to detect[16][11]. Mutation testing has been studied since 1977. Mutation testing is based on the Competent Programmer Hypothesis and the Coupling Effect Hypothesis[11]. The Competent Programmer Hypothesis assumes that programmers are competent and write programs that are close to being correct. A correct program can be created from an incorrect program that includes syntactically small faults and with a few small code modifications. The

Coupling Effect Hypothesis indicates that test cases that distinguish all programs differing from a correct one by only simple errors are so sensitive that they can distinguish programs with more complex differences. So mutation testing can be used to simulate complex real-world bugs, especially for bugs that are hard to detect and reproduce.

2.3 Mutation Testing Tool

Without a fully automated mutation testing tool, creating mutants can be a cumbersome process, especially for large programs. Therefore, the development of mutation testing tools is necessary. Various mutation testing tools have been developed. MuJava[27] is a mutation tool for Java that includes class-level operators. MOTHRA[12] is a mutation testing tool for Fortran. MILU[21] is an efficient and flexible mutation testing tool designed for both first order and higher order mutation testing in C. Jester is the first open source mutation testing tool for Java. Its two mutation operators are very similar; one changes 0 to 1 and the other replaces predicates with TRUE and FALSE[18]. Pester[18] is a Python version of Jester. Nester[1] is an open source tool for C Sharp. Moreover, there are several mutation tools like INSURE++[30], PLEXTEST[20], CERTITUDE[9] available commercially.

2.4 Eclipse

Eclipse is an integrated development environment (IDE). It is written mostly in Java. Typically, it consists of a base workspace and an extensible plug-in system for customizing the environment[2]. Plug-ins can be used to build arbitrary applications in different programming languages under different development environments. In

other words, everyone can contribute plug-ins and Eclipse can use its strong extensible plug-in system to integrate various features in a single working platform.

2.4.1 Architecture of Eclipse Plug-ins

Eclipse is not just a single working platform, but rather a small kernel with a plug-in loader surrounded by thousands of plug-ins. The small kernel is based on a container that is implemented by OSGi R4 and provides the environment to control the plug-ins execution[10]. Each plug-in contributes itself in a structured manner, may be based on services provided by another plug-in and each may in turn provide services on which other plug-ins may rely. An Eclipse plug-in, typically, consists of two components, extensions and extension points, respectively. The concept of extensions and extension points allow functionality to be contributed to plug-ins by other plug-ins(see Figure 2.2).

2.4.1.1 Extension Points

When a plug-in wants to allow other plug-ins to extend portions of its functionality, it declares an extension point. The extension point declares a contract, typically, a combination of XML markup and Java interfaces, that extensions must conform to[2]. Plug-ins must implement that contract in their extension if they want to plug in to that extension point.

2.4.1.2 Extensions

Extensions are plug-ins which contribute an extension. Typically, these plug-ins provide an extension based on the contract that was defined by a corresponding extension point. Extensions can be either code or data (see Figure 2.3).

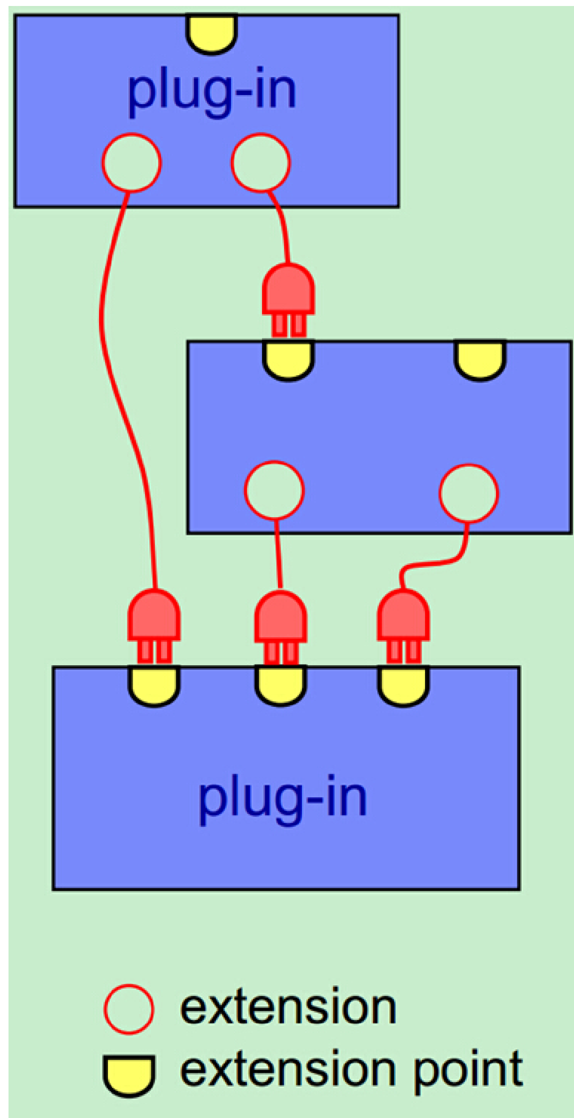

Figure 2.2: Eclipse plug-ins

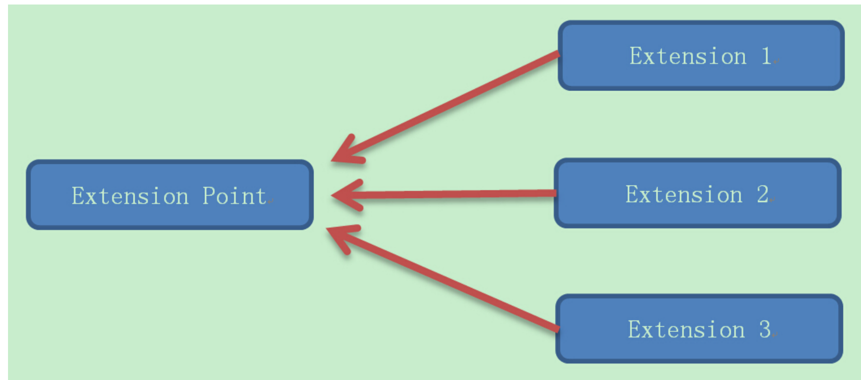

Figure 2.3: Extensions and extension points

2.4.2 C/C++ Development Tooling (CDT)

As we mentioned, in Eclipse everything is a contribution (plug-in). Because of its strong extensible plug-in system, Eclipse is not only an IDE for Java programming, but also an IDE for other popular programming languages like C++ and PHP. When Eclipse was used only as a Java programming IDE, the development tooling in Eclipse was Java development tooling (JDT). When Eclipse became a general application platform, each programming language provided its own corresponding development tooling. For C/C++, C/C++ development tooling (CDT) is an Eclipse plug-in that transforms Eclipse into a powerful C/C++ IDE. It can offer many of the features Eclipse provides to Java developers to C/C++ developers. Basically, the core of CDT consists of a preprocessor, parsers (C/C++), an abstract syntax tree (AST), an AST rewrite API, semantic analysis (name resolution), an indexer and an Index API. The tool we create in this work is not development tooling or a compiler, so we rely on only three core parts of CDT, a preprocessor that converts text into a token stream, parsers (C/C++) that convert the token stream into an AST and an abstract syntax tree (AST) representation of the syntactic structure of source code written in C/C++.

2.4.2.1 Visitor Pattern API for ASTs

An abstract syntax tree (AST) is a tree representation of abstract syntactic structure of source code written in a programming language[3]. Basically, an AST is used for semantic analysis where the compiler checks whether the element of the programming language is correctly utilized. However, traversing an AST is not an easy job. The problem here is that the type of each node is different. For example, the AST of $a = b + c$ has three different nodes, an assignment operator, a variable id and an arithmetic operator. Since each node may correspond to a class, the AST traversal may go through all the classes, which makes the program hard to read and maintain. The solution to this problem is to utilize a design pattern called the visitor pattern instead of sifting through all the classes. The visitor pattern lets us traverse the AST using different visitors. More accurately, each node of the AST has an accept method accepting a call from a visitor that performs its custom traversal. So we can use the visitor pattern to traverse a particular block, statement, expression or declaration in a source file.

Chapter 3

Design and Implementation

3.1 Fault Patterns

In this section, we present a set of fault patterns designed as mutants, with which to seed a healthy C/C++ program. First, we created a concurrency fault taxonomy to identify the reasons for the most common concurrency faults. We used the ROS[4] bug repository as a resource to do this. ROS stands for Robot Operating System, and is a flexible framework for writing robotics software. To collect the most common concurrency faults, we used the terms deadlock, synchronization, mutex and race condition as keywords to query for faults related to concurrency. Table 3.1 presents data on keywords and real faults. Figure 3.1 presents the real reasons these faults occur. We can see that most concurrency faults are associated with lock() or unlock() methods. Figure 3.2 represents the most common fault types. We found that most faults can generate deadlock or race conditions. So according to the data we collected, we designed four types of mutant operators. They are Remove Unlock, Remove Lock, Remove Paired Lock and Unlock and Switch Lock Order, respectively.

Keywords	*Deadlock*	*Race Condition*	*Synchronization*	*Mutex*	*Multiple Thread*	*Simultaneous*	***Total***
Keywords Containing	91	246	72	189	62	55	715
Related	5	15	3	3	6	2	34

Table 3.1: Concurrency Fault Taxonomy

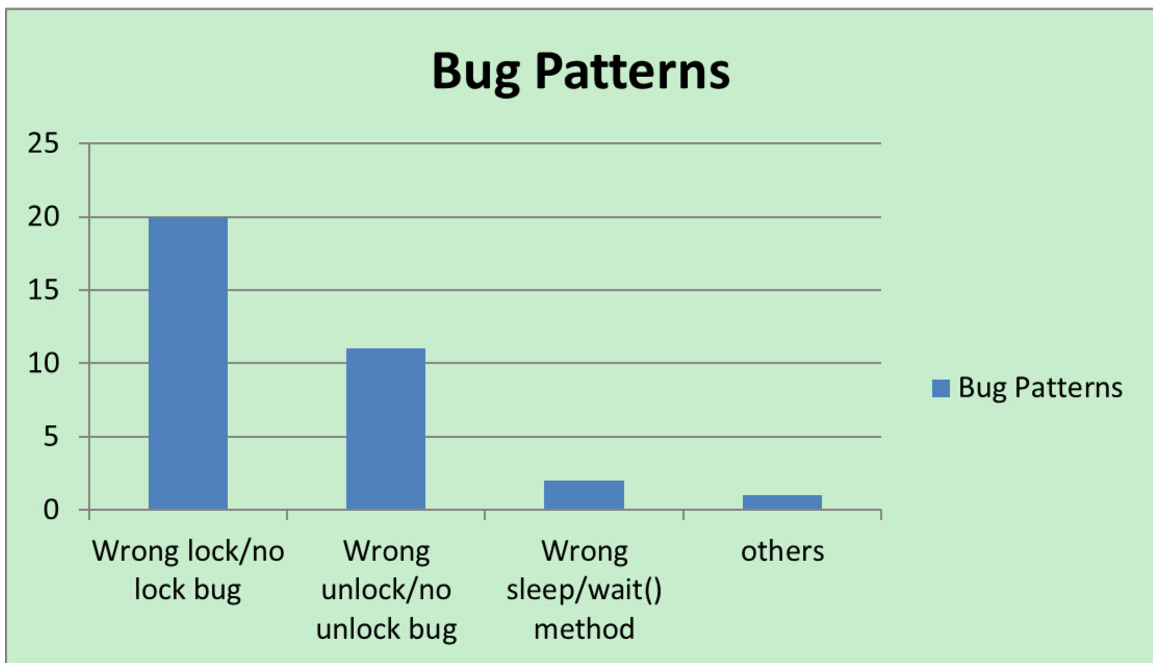

Figure 3.1: Bug patterns

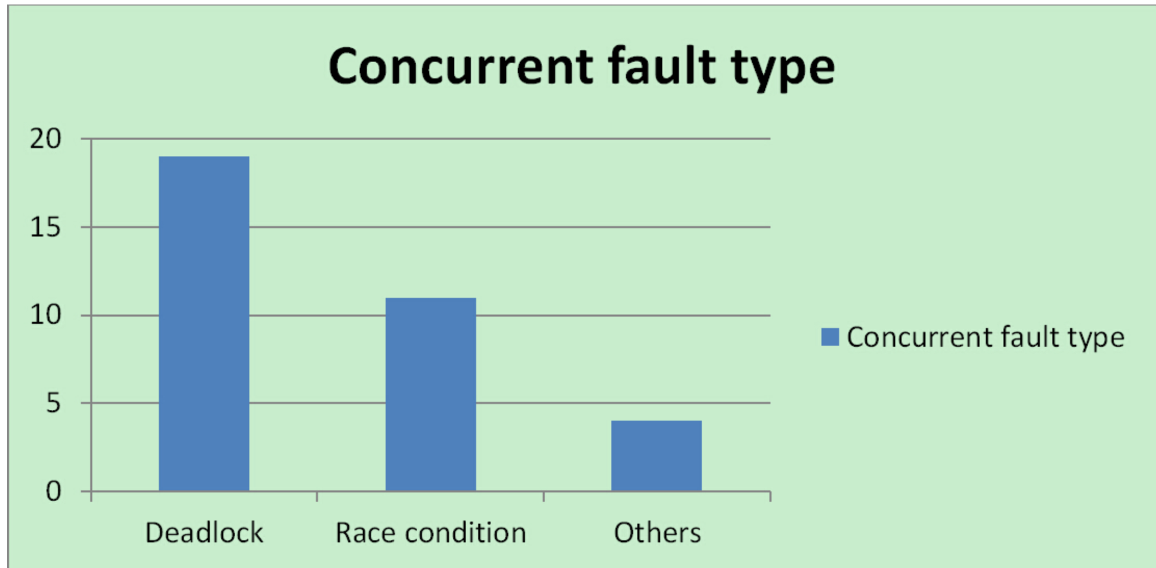

Figure 3.2: Concurrent fault types

3.1.1 Remove Unlock

Improper use of unlock or missing unlock faults are very common in concurrent programs. This type of fault occurs when developers do not use unlock() functions properly. For example, an unlock() may not be paired with its lock() in cases where interactions among threads are complicated. Meanwhile, this type of fault can cause deadlock. The Remove Unlock operator is the mutant used to delete one unlock method in concurrent programs to simulate a fault due to a missing unlock. Program A provides a simple example of this type of fault.

```
  P1{
1.  Lock(mutex);
2.  x++;
3.  ...
4.  //Unlock(mutex); //fault
  }
```

Program A

3.1.2 Remove Lock

Incorrect or missing locks are another very common type of fault in concurrent programs. This type of fault occurs due to improper use of lock operations in a program that may require multiple locks to be managed. The Remove Lock operator is the mutant used to delete locks in concurrent programs to simulate missing lock faults. Program B provides a simple example of this type of fault.

```
  P2{
1.  //Lock1() //fault
2.   ...
3.  Lock2()
4.
5.  Lock3()
6.   ...
 }
```

Program B

3.1.3 Remove Paired Lock and Unlock (Critical Section Violation)

Critical section violations are a common faults in concurrent programs. This type of fault occurs if a critical section is not protected properly, allowing it to be accessed by multiple threads at one time. Typically, this type of fault is the main reason for critical section violations. The Paired Lock and Unlock operator is the mutant used to delete paired lock and unlock methods in the same block in concurrent programs to simulate faults due to critical section violations. Program C provides a simple example of this type of fault.

```
P3{
1.  //Lock(); //fault
2.  x++;
3.  ...
4.  //Unlock();//fault
}
```

Program C

3.1.4 Switch Lock Order

Incorrect lock order is another cause of concurrency faults in concurrent programs. This type of fault occurs due to improper use of lock operations in programs that require multiple locks to be managed. The Switch Lock Order operator is the mutant used to change the lock order in the same block in a concurrent program, to simulate this class of fault. Program D provides a simple example of this type of fault. M_P4 represents the program after injecting a mutant.

```
P4{
1.  Lock1();
2.  Lock2();
3.  ...
4.  Unlock2();
5.  Unlock1();
}

M_P4{
1.  Lock2();// fault
2.  Lock1();// fault
3.  ...
```

```
4.   Unlock2();
5.   Unlock1();
  }
```

Program D

3.2 Implementation of a Concurrency Fault Injection Tool

The Concurrent Fault Injection Tool (CFIT) is our concurrency fault mutation system for the C/C++ programming languages. It automatically generates mutants for concurrent mutation testing based on the aforementioned fault patterns. CFIT is developed as an Eclipse plug-in. It can analyze single C/C++ source files or a whole C/C++ project. Mutants of a C/C++ file are generated inside conditional compilation constructs in the original source file and activated via an automatically generated mutant header file.

3.2.1 CFIT Architecture

CFIT consists of four components: Injection Action Extension, Mutation System, Mutant Property, and Database.

3.2.1.1 Injection Action Extension

The Injection Action Extension is a module that performs fault seeding. Its main GUI is in the form of a pop-up menu. It is an extension connecting to a particular extension point, org.eclipse.ui.popupMenus. This extension point is used to add new actions to context menus defined by other plug-ins. To use this plug-in, the user only

needs to right click the project that is the target for injected faults. Next, on the pop-up menu, the user selects the fault injection option. Mutants will be injected automatically and the mutant source file and mutant header file will automatically be generated in a user specified path (see Figure 3.6).

3.2.1.2 Mutation System

The Mutation System is the core component of CFIT. It consists of three parts: CDT parser, abstract syntax tree (AST) and mutant property.

CDT is implemented in the C/C++ development tooling. Because CDT has a full C/C++ parser and AST, we decided to use the CDT parser and AST directly. The CDT parser is the component used to parse C/C++ source code. It takes a C/C++ program as input and parses the source into a token list. The token list, typically, will generate an abstract syntax tree. However, because the official CDT does not let the user access the AST, we downloaded a developed version of the CDT package which includes a test mode that lets the developer use a DOM AST component and a debugging component.

The main package for the AST for C/C++ is called org.eclipse.cdt.ui.tests.DOMAST. It is located as a sub-project of CDT called org.eclipse.cdt.ui.tests. This package is mainly used for traversing an AST in the form of a GUI so that the CDT developer can retrieve the ASTNode information during development. Each C/C++ source file is represented as a subclass of the ASTNode class. Each specific AST node provides specific information about the object it represents. To traverse an AST and obtain ASTNode specific information, we use the visitor pattern. This lets us write user defined plug-ins that process the AST. We built subclasses based on the visitor pattern extending the ASTVisitor class (Figure 3.3), which is an abstract base class to which visitors can traverse AST nodes and override methods that users specify

for different subclasses. Moreover, because the CDT DOM AST has its own built-in node classes that each has an accept (ASTVisitor) method, we do not need to build these accept methods by ourselves. In other words, we only need to create a visitor object extending ASTVisitor and override several overloaded visit methods for each node type, and then we can process the AST in forms that we want.

Figure 3.4 provides an example, showing a subclass of ASTVisitor. The LockManagementVisitor class is used to obtain all lock methods in one IASTTranslationUnit and their AST node-specific information in a single C/C++ source file. IASTTranslationUnit is a compilable unit of source. Typically, we consider it to be the root of an AST. It accepts a user defined visitor class (e.g. LockManagementVisitor) and processes a particular traversal based on several overridden visitor methods. Since we can get ASTNode information such as line number, parent ASTNode, children ASTNodes, etc. in a source file, we can operate on any statements, expressions, or variables in any desired manner. For example, if we want to remove one specific lock method in a specific compound statement, we only need to get this specific lock method's ASTNode information based on a user-defined visit method in a specific subclass that extends the ASTVisitor class. Then according to the ASTNode's specific information, we can easily locate that lock method in a source file and insert the conditional compilation directives that implement the mutation using specific string operations.

3.2.1.3 Mutant Property

The Mutant Property is the component used to retrieve user specified mutant operators as the input for the mutation system. Right now, as described earlier, we have four mutant operators: Remove Unlock, Remove Lock, Switch Lock Order and Remove Paired Lock and Unlock. We use Java properties file format to set up the

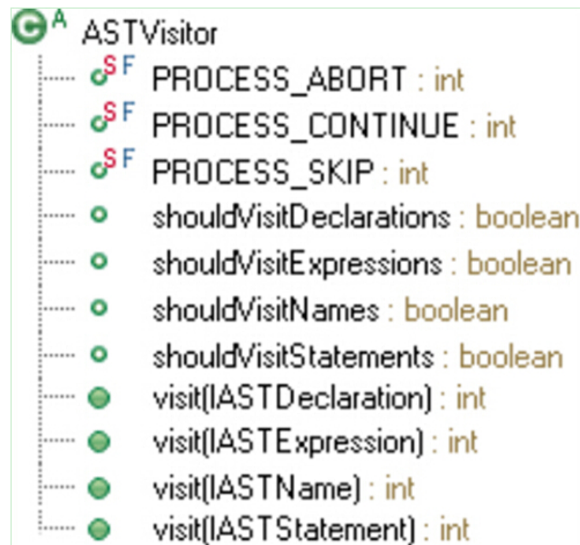

Figure 3.3: Class ASTVisitor in DOM AST

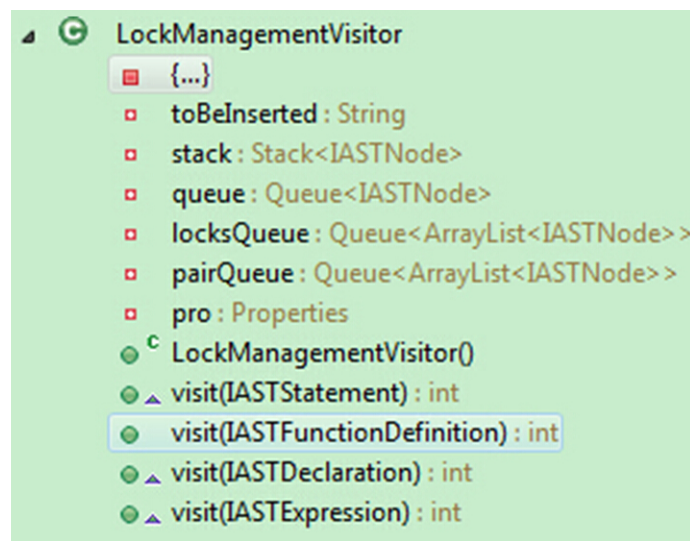

Figure 3.4: Class LockManagementVisitor

rules for mutants. If we want to open a mutant operator, we set the property value to “yes”. If not, we set the property value to “no”. For example, RemoveUnlock=yes tells the mutation system to activate the remove unlock pattern during runtime. Each time, we seed only one type of mutant: if one mutant operator is opened, the other three must be closed.

The mutant template is another Java properties file that is used to obtain lock or unlock information in an application. For example, if we want to seed a Remove Unlock pattern in an application, we need to specify the unlock method name in the mutant template. For example, Unlocker=pthread_mutex_unlock represents the case of a Remove Unlock pattern opening in which the mutant system will seed the mutant only when the unlock method name of the specific application is pthread_mutex_unlock.

3.2.1.4 Database

Due to the large number of mutants generated by CFIT, we use a database to conveniently track each mutant's specific information, including the name of the injected source file, the fault pattern, and the location of the mutant (line number).

3.2.1.5 Hibernate

Because our database is designed with respect to an object relational mapping model, we chose the Hibernate ORM as our database framework. However, due to the way Eclipse RCP (and Eclipse in general) delegates class loading to buddy plug-ins[5], it is necessary to wrap third-party libraries in a plug-in to ensure that the correct context class loading occurs at runtime. Hibernate is an open source software providing a framework with which to map an object oriented model to a traditional relational database[7]. Because Hibernate is a third party library for Eclipse RCP, importing Hibernate into a single Eclipse plug-in project will not activate the database. To solve

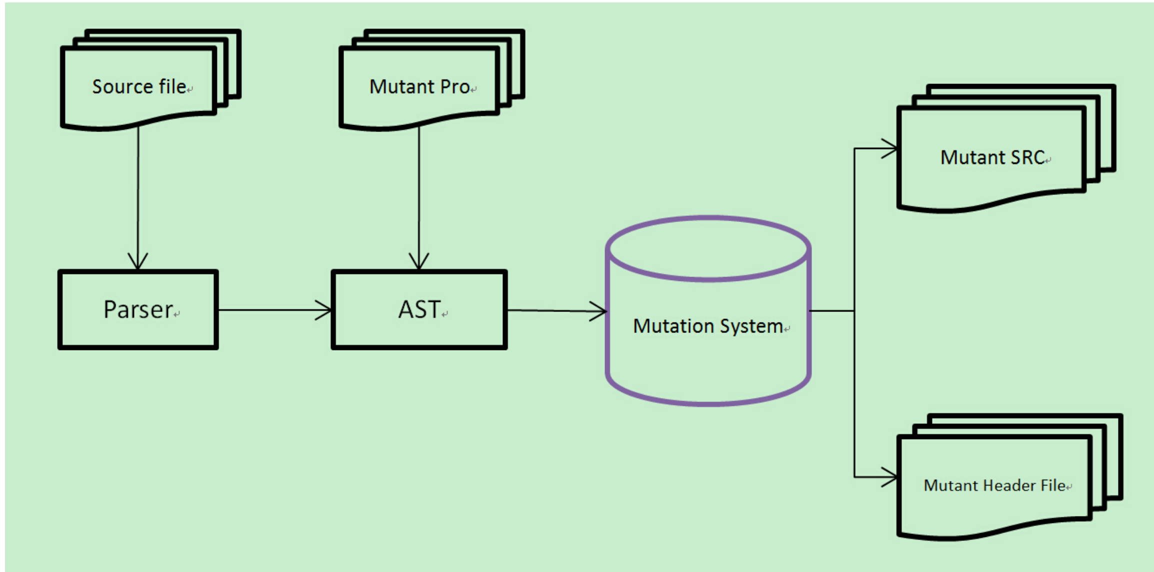

Figure 3.5: CFIT procedure

this problem, we built another plug-in project just for the database part, imported all the libraries, and used this standalone plug-in as a dependency of CFIT. Then the database can be active during the CFIT run-time.

3.2.1.6 CFIT Working Process

Figure 3.5 represents the working process of CFIT. The mutation system takes the AST from the CDT parser and the mutant property that a user has defined as input and generates a mutant in the form of conditional compilation in the source code and a mutant header file as the mutant switch. Each mutant is represented in the form “FaultMutantPattern_MutantId”. For example, if mutant operator properties activate the Remove Unlock pattern and the mutant template sets unlock to pthread_mutex_unlock, Fault_Remove_Unlock_m0 will be generated in the form of conditional compilation. FaultRemoveUnlock_m0 indicates that the mutant removes one unlock method in the source and its id is listed as 0.

As an example, Program E illustrates how a mutant is generated in a source file.

At the same time, a mutant switch corresponding to that mutant is generated in the mutant header file "sourceFileName_mutant.h". Each mutant header file contains a certain quantity of mutants starting with two slashes that can also be considered as a comment in a regular program. Each mutant is represented in the form of a #define directive that defines a constant and creates a macro. If a mutant needs to be active, we only need to remove the two slashes and then the mutant will switch from comment to macro.

Program F is a simple example showing how a mutant header file works. We combine program E and program F to show how a mutant is activated. In program F, when we remove two slashes from the first line, #define FAULT_unlock_remove_m0 0 will be activated from comment status. At this point, FAULT_unlock_remove_m0 is defined and the constant value of this definition is 0. In other words, it is defined. Returning to program E, line 1 represents whether FAULT_unlock_remove_m0 is defined, so the routine will go to line 2 that does nothing, omitting the call to the pthread_mutex_unlock(mutex) method, and then go to line 5 and continue. If we switch the first line of the mutant header file from macro status to comment again, the mutant FAULT_unlock_remove_m0 will be closed, and then if we rerun program E, line 4 will be executed. So we can see that when one type of mutant is set, all feasible mutants will be seeded in the source file in the form of conditional compilation and be listed in the mutant header file. It is convenient to open and close a mutant by just deleting two slashes or adding two slashes back.

```
  P5{
   ...
1.  #ifdef FAULT_unlock_remove_m0
2.
3.  #else
4.      pthread_mutex_unlock(mutex);
```

```
5.  #endif
  ...
  }
```

Program E

```
//#define FAULT_unlock_remove_m0 0
//#define FAULT_unlock_remove_m1 0
//#define FAULT_unlock_remove_m2 0
  ...
```

Program F

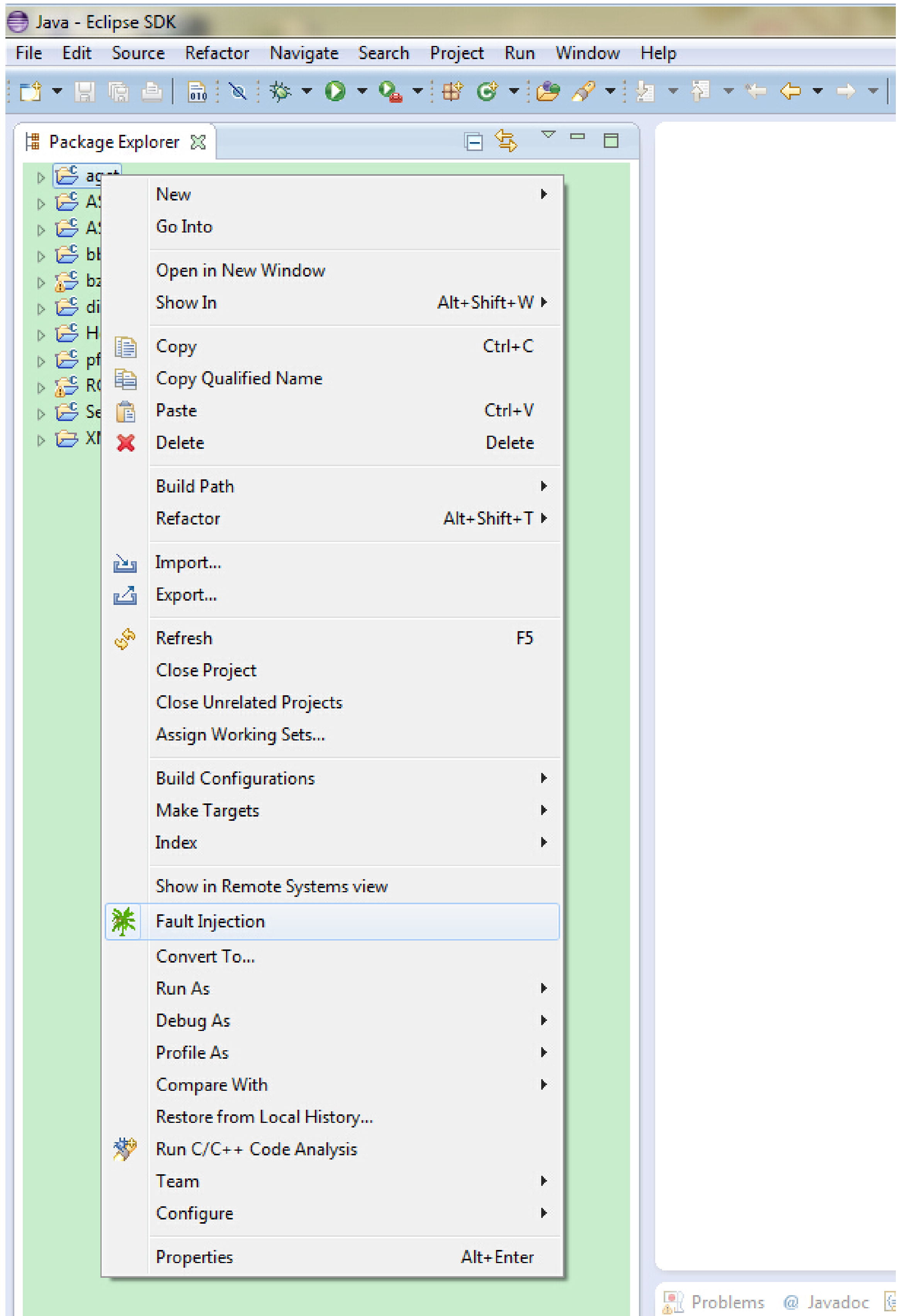

Figure 3.6: Snapshot of programs after modifications made by CFIT

Chapter 4

Empirical Study

In this chapter, we provide an empirical evaluation of the proposed framework. We focus on its efficiency and ability to generate challenging mutants that can be helpful in studying techniques for uncovering difficult to detect concurrency faults.

4.1 Purpose of Study

The purpose of this study is to evaluate the feasibility of the approach of using an automation injection tool instead of manually injecting concurrency faults in studies of testing, and assess the efficiency of mutant generation and characteristics of the mutants that can be exposed. We consider the following research questions:

RQ1: Whether and to what extent are mutants generated by CFIT detectable?

RQ2: Are the mutants not too easily detectable?

RQ3: Is our tool efficient enough?

4.2 Objects of Study

To evaluate our tool and methodology, we chose five concurrent programs. They include BBUF, which is an implementation of the producer and consumer program, AGET, which is a multithreaded FTP download application, PFSCAN, which is a parallel file scanner, BZIP[6], which is a multithreaded compression program, and DININGPHILOSOPHER, which is an example from the Oracle Thread Analyzer[29]. The reason we select these programs for our study is because they include real-world programs, commonly used concurrency benchmarks and commercial tools. Furthermore, these applications have been used in prior studies of techniques for testing for concurrency faults[39].

Because our object programs are not distributed with test cases, we needed to generate test cases for them. We consider three factors in generating test cases: test input data, other relevant parameters, and specified thread execution interleavings[23]. Before we generate a large number of test cases, we need to consider output files based on the four mutant operators used by CFIT. Each injected program contains a corresponding mutant specification in the form of a header file. It specifies mutants that have been injected into the program and supports the ability to enable one particular mutant through a mutant generator program. For example, if one mutant header file includes 8 mutants, after running the mutant generator program, 8 different versions of that program will be generated. For each version of the program, we created a set of valid test input values and command options with different numbers of threads ranging from 1 to 5.

For each of these test inputs, we assigned a thread interleaving by randomly selecting a set of program locations at the granularity of instructions. We randomly added yield points to these selected locations; this has a high probability of achieving

determinism[23]. A yield point is used to make a thread voluntarily suspend its execution. This creates an environment where *interleavings happen more frequently and under much greater control by the tester.* This is accomplished by injecting sleep functions for a finite amount of time so that the scheduler would pick other threads to run. This allows a tester to control thread interleaving.

Program G provides a simple example of how a yield point works. Between lines 2 and 3, a sleep function call is inserted as the yield point to cause the current thread (*thread A*) to suspend execution for one second. Thus, another thread (*thread B*) is scheduled while *thread A* is sleeping, resulting in a controlled interleaving. To further explore different interleaving patterns at runtime, we generated 10 test cases with different yield points for each mutant.

```
...
1. movl (count), %eax;
2. addl $1, %eax;
  sleep(1000); // Yield point
3. movl %eax, (count);
...
```

Program G

The end result of this process is a relatively large number of test cases. For example, if we have 8 mutants for the RemoveUnlock pattern, there will be 8 * 10 * 5 = 400 executions, where 8 is the number of mutants, 10 is the number of test inputs used for each mutant and the number of threads ranges from 1 to 5. Moreover, the number 400 here is just for one type of mutant operator; if each of 4 mutant operators can generate 400 test cases, there would be 400 * 4 = 1600 unique test cases generated for that program.

4.3 Study Operation

Figure 4.1 illustrates the process we used to generate and execute test cases on all of the faulty versions of each object program, with one mutant activated on each execution. The reason we activate only one mutant in each execution is to avoid fault-interactions and masking effects, and to allow us to accurately determine whether each mutant was indeed detected. The basic procedure is as follows: (1) CFIT generates a number of mutant files including mutant source files and corresponding mutant header files. (2) A mutant generator opens each mutant header file and then generates a new version of the program based on each specified mutant; these are then compiled. (3) TC Gen is the test case generation tool. Each test case consists of different yield point files generated by the yield point generator (YP Gen), a set of test inputs, a number of threads ranging from 1 to 5, and various command options for each object program based on different types of mutants. (4) We use Pin, which is a dynamic binary instrumentation tool,[19] to execute the test cases. (5) We then employ an algorithm based on wait-for graphs [34] to detect deadlocks. If a circular-wait condition is detected, the deadlock detector reports the program, the specific test case, the specific mutant, and the number of threads that result in that particular deadlock. Moreover, the system also creates an event log after each execution that can be used for further analysis.

4.4 Result

Table 4.1 lists our five concurrent programs and data on their mutants. Column 1 is the name of the program. Numbers of lines of code is listed in Column 2. Columns

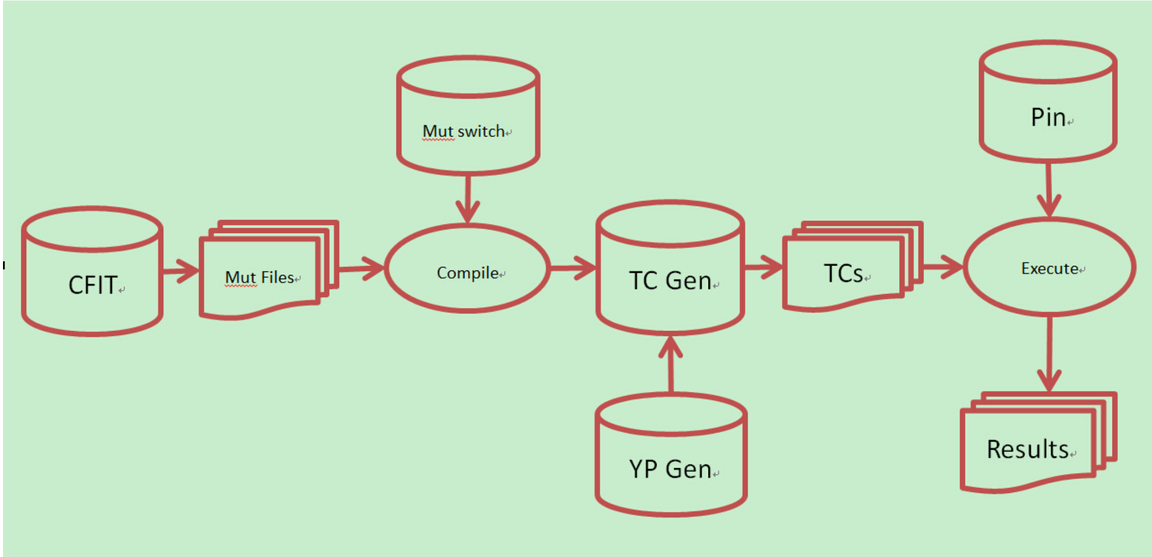

Figure 4.1: Experiment procedure

3 to 6 report the numbers of mutants generated based on each mutant operator.

Program	NLOC	Rm Unlock	Rm lock	Rm paired Lock and Unlock	Switch locks order
BBUF	256	8	6	6	0
DIN.PHIL	104	5	4	3	0
AGET	846	2	2	2	0
PFSCAN	752	12	11	10	1
BZIP	4232	10	11	9	142

Table 4.1: Mutant Data

Tables 4.2–4.5 report results regarding the effectiveness of the proposed framework in creating challenging but detectable mutants. Each table represents a particular fault pattern. In each table, Column 1 provides the name of the object program. Column 2 provides the number of mutants of that mutant operator generated by CFIT. Column 3 denotes how many deadlocks are detected after executing the mutants. Column 4 is the report of the mutation score for that type of mutant. The

mutation score is based on the ratio between the percentage of detected and injected mutants.

Program	Rm Unlock	DLs Detected	Mutation Score
BBUF	8	6	75%
DIN.PHIL	5	5	100%
AGET	2	2	100%
PFSCAN	12	5	40%
BZIP	10	4	40%

Table 4.2: Remove Unlock

Table 4.2 is the result of the RemoveUnlock mutant operator. We can see that deadlocks occur in all of the programs (see Column 3). However, except for on programs DIN.PHILO and AGET, not all of the mutants are detected or killed. The mutation score for BBUF is 75%, and for both PFSCAN and BZIP it is 40%.

Program	Rm lock	DLs Detected	Mutation Score
BBUF	6	0	0%
DIN.PHIL	4	2	50%
AGET	2	0	0%
PFSCAN	11	3	27%
BZIP	11	0	0%

Table 4.3: Remove Lock

Table 4.5 reports the results when we apply the Switch Lock Order mutant operator. Note that BBUF, DIN.PHILO and AGET do not have mutants of this type. In these three applications, there is only one lock statement in each block. For PFSCAN, only one mutant is generated and it is killed by our test cases. For BZIP, 94 mutants are killed out of 142.

Next we describe the results reported in Table 4.3 and Table 4.4. We can see that both the Remove Lock and Remove Paired Lock and Unlock mutant operators did

not cause deadlock to occur in BBUF, AGET and BZIP. Therefore, the experiment results show that two mutant operators, Remove Unlock and Switch Lock Order, can cause deadlock to occur more easily than mutant operators Remove Lock and Remove Paired Lock and Unlock. The reason for this is that removing an unlock can cause a thread to exclusively hold a resource without releasing it, resulting in circular waits. We also find that switching the order of two locks often results in circular wait. Although the mutants generated by Remove Lock and Remove Paired Lock and Unlock are hard to kill, deadlocks still occur during run time. The reason for this is that these two mutant operators can easily cause data races and data races are a potential factor that can lead to deadlock.

Program	Rm lock	DLs Detected	Mutation Score
BBUF	6	0	0%
DIN.PHIL	3	3	100%
AGET	2	0	0%
PFSCAN	10	1	10%
BZIP	9	0	0%

Table 4.4: Remove Paired Lock and Unlock

Program	Rm lock	DLs Detected	Mutation Score
BBUF	0	-	-
DIN.PHIL	0	-	-
AGET	0	-	-
PFSCAN	1	1	100%
BZIP	142	94	66%

Table 4.5: Switch Lock Order

To further evaluate our mutation approach, Table 4.6 lists the total numbers of detected mutants based on all 4 mutant operators and the number of detected mutants based on the percentage of test cases. For example, if a mutant is detected by all test cases, then it would be reported in the last column (80% - 100%). Due to the need to

produce meaningful results in experiments on testing techniques, seeded faults should be neither too easy nor too hard to detect[17]. If they are too hard to detect, then all mutants are not likely to be killed by any test cases, and they provide no ability to differentiate approaches. (Note, however, that some of mutants that cannot be killed may actually be equivalent mutants, which are mutants that behave equivalent to the base program.) Conversely, if mutants are too easy to detect, then almost any test cases can detect them, and they are likely to be detected by any testing technique, again providing no ability to differentiate approaches.

Program	NMs	0.1-20%	20-40%	40-60%	60-80%	80-100%
BBUF	6	1	0	1	1	3
DIN.PHIL	10	0	0	0	0	10
AGET	2	0	0	0	0	2
PFSCAN	10	8	0	0	0	2
BZIP	98	34	3	9	1	51

Table 4.6: The total numbers of detected mutants based on all 4 mutant operators and the number of detected mutants based on the percentage of test cases

We now turn to our research questions. We first consider whether mutants are detectable (RQ1). Based on results reported in Tables 4.2 through 4.5, we can see that only in the cases of BBUF and AGET under the Remove Lock and Remove Paired Lock and Unlock patterns did mutants fail to cause deadlock to occur. In the remaining three programs injected mutants did cause deadlocks to occur. Although mutants under Remove Lock and Remove Paired Lock and Unlock for BBUF and AGET are not killed, we have not determined whether these are equivalent mutants or whether the test cases are not adequately constructed to reach them. We leave this analysis as future work. In summary, our experiment results show that a large proportion of the mutants generated by CFIT are detectable.

We now consider whether our mutants are not too easily detectable (RQ2). Table

4.6 reports that around 34% of all mutants fall in the category of 0.1-20% detection ratio. In this category, mutants are detectable but only by some test cases. As such, these mutants are detectable but detecting them can be challenging[1]. However, on DIN.PHILO and AGET, results are not encouraging, with all mutants being easily detected. We believe this is due to the problem discussed earlier, namely, most of the test cases for these programs are not strong enough to detect more difficult-to-detect mutants. Like RQ1, we will further investigate this issue as part of future work.

Finally, we consider whether our tool can operate efficiently (RQ3). As a preliminary evaluation, we measured the amount of time needed to inject 244 faults across all four fault patterns in all five applications. The amount of time needed was around 5 minutes. On the other hand, manual injection would likely take longer to perform the same task. As such, we conclude that the proposed CFIT is efficient.

Program	DLs Detected
BBUF	F
DIN.PHIL	T
AGET	F
PFSCAN	F
BZIP	F

Table 4.7: Deadlocks for Base Program

4.5 Discussion

Before we discuss our results, we also ran another experiment using the same test cases to run the base programs. The results are shown in Table 4.7. We can see that deadlock previously exists in DININGPHILOSOPHER. The other four applications have no detectable deadlocks prior to apply CFIT.

[1] We used increment of 20% as previously used by [32].

We now discuss the results of our empirical study. With respect to mutation score, the results show that DININGPHILOSOPHER has the highest score across all fault patterns except for the Switch Lock Order pattern. As a reminder, the Switch Lock Order pattern is not applicable to this program. The scores for Remove Unlock and Remove Paired Lock and Unlock are 100%. For Remove Lock, the score is only 50% but yet, this score is still the highest when compared to those of the remaining four programs. According to the result showing in Table 4.6, we can see that all killed mutants are located in the 80–100% category, implying that all mutants generated by CFIT for DININGPHILOSOPHER are easily detectable.

DININGPHILOSOPHER was released by Oracle as a test program for its tool, Thread Analyzer. This particular tool can be used to analyze the execution of a multithreaded program. Typically, it can detect multithreaded programming errors such as data races and deadlocks in code that is written using the POSIX thread API, the Solaris thread API, OpenMP directives, or a mix of these[29]. As a test program, it already contain sources of deadlock before we injected it with mutants. As such, adding more mutants causes deadlock to occur even more easily, which is reflected in its high mutation score.

Next, we analyzed the mutation score of the remaining 4 programs. For Remove Lock and Remove Paired Lock and Unlock, only PFSCAN has mutants 4 out of 21 that can cause deadlock to occur. The remaining three programs do not have any mutants that can cause deadlock to occur. As we mentioned above, the reason most of mutants are not killed by our test cases is that some may actually be equivalent mutants. Other non-equivalent mutants may fail to be killed may be due to inadequate test cases.

For the Remove Lock or Remove Paired Lock and Unlock, we basically remove protection from critical sections. This can result in data races. It is quite possible

that data races can lead to deadlocks. We found 4 mutants that can cause deadlocks due to races.

Finally, we analyzed the mutation scores of Remove Unlock and Switch Lock Order patterns. Missing corresponding unlocks or incorrect lock orders are major fault patterns that can cause deadlocks in concurrent programs. This is because missing unlock operations can result in more mutually exclusive resources. Mutual exclusion is an important factor that can lead to deadlocks. Switching lock orders can also lead to more hold and wait instances in nested locking situations. According to Table 4.2 and Table 4.5, the experiment results indicate that mutants based on these two patterns are likely to cause deadlocks.

Chapter 5

Conclusion and Future Work

In this paper, we have presented a methodology for injecting mutations related to concurrency faults. We built an automatic concurrent fault injection tool (CFIT) based on an Eclipse plug-in for C/C++. In an empirical study, we evaluated our tool's effectiveness by using it to seed various types of concurrency faults based on four fault patterns into five concurrent programs. Our results show that using the proposed concurrent fault injection tool (CFIT) is feasible as a basis for empirically evaluating testing techniques.

In future work, we intend to incorporate more mutant operators into CFIT such as Shift Critical Section and Modify Mutex. We also intend to extend our study of internal oracles to take other concurrency faults into account such as critical section violations and starvation. Finally, we intend to perform more empirical studies to evaluate the effect of equivalent mutants and non-equivalent mutants that are not killed in our work.

Bibliography

[1] http://nester.sourceforge.net.

[2] https://www.eclipse.org/.

[3] http://en.wikipedia.org/wiki/Abstract_syntax_tree.

[4] http://www.ros.org/.

[5] http://wiki.eclipse.org/index.php/Context_Class_Loader_Enhancements.

[6] http://bzip2smp.sourceforge.net/.

[7] Hibernate. http://hibernate.org/.

[8] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[9] Cetress. Certitude. http://www.certess.com/product/.

[10] E. Clayberg and D. Rubel. *Eclipse Plug-ins*. Addison-Wesley Professional, 3 edition, 2008.

[11] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[12] R.A DeMillo, D.S. Guindi, W.M. McCracken, AJ. Offutt, and K. N. King. An extended overview of the mothra software testing environment. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 142–151, Jul 1988.

[13] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. *SIGPLAN Not.*, 35(5):219–232, May 2000.

[14] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, May 2003.

[15] Sudipto Ghosh. Towards measurement of testability of concurrent object-oriented programs using fault insertion: A preliminary investigation. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, pages 17–26, Washington, DC, USA, 2002. IEEE Computer Society.

[16] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, July 1977.

[17] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[18] I.Moore. http://jester.sourceforge.net.

[19] Intel. Pin. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[20] Itregister. Plextest. http://www.itregister.com.au/products/plextest.htm.

[21] Yue Jia and Mark Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques*, TAIC-PART '08, pages 94–98, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, September 2011.

[23] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag.

[24] B. Long, R. Duke, D. Goldson, P. Strooper, and L. Wildman. Mutation-based exploration of a method for verifying concurrent java components. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 265–, April 2004.

[25] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. *SIGOPS Oper. Syst. Rev.*, 40(5):37–48, October 2006.

[26] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 553–563, New York, NY, USA, 2009. ACM.

[27] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: An automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, June 2005.

[28] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, June 2007.

[29] Oracle. Thread analyzer. http://docs.oracle.com/cd/E18659_01/html/821-2124/gepdy.html.

[30] Parasoft. Parasoft insure++. http://parasoft.com/jsp/products/home.jsp?product=Insure.

[31] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: Fault localization in concurrent programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 245–254, New York, NY, USA, 2010. ACM.

[32] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[33] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems.*, 15(4):391–411, November 1997.

[34] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Applied Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[35] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sergio Lopes de Souza. Mutation operators for concurrent programs in mpi. In

Proceedings of the 2012 13th Latin American Test Workshop - LATW, LATW '12, pages 1–6, Washington, DC, USA, 2012. IEEE Computer Society.

[36] William Stallings. *Operating Systems - Internals and Design Principles (7th ed.)*. Pitman, 2011.

[37] Aditya Thakur, Rathijit Sen, Ben Liblit, and Shan Lu. Cooperative crug isolation. In *Proceedings of the Seventh International Workshop on Dynamic Analysis*, WODA '09, pages 35–41, New York, NY, USA, 2009. ACM.

[38] Leon Li Wu and Gail E Kaiser. Empirical study of concurrency mutation operators for java. 2010.

[39] Tingting Yu, W. Srisa-an, and G. Rothermel. An empirical comparison of the fault-detection capabilities of internal oracles. In *2013 IEEE 24th International Symposium, Software Reliability Engineering (ISSRE) on*, pages 11–20, Nov 2013.