Constructing Subtle Concurrency Bugs Using Synchronization-Centric Second-Order Mutation Operators

Leon Wu Gail Kaiser
Department of Computer Science
Columbia University
New York, NY 10027 USA
{leon,kaiser}@cs.columbia.edu

Abstract

Mutation testing applies mutation operators to modify program source code or byte code in small ways, and then runs these modified programs (i.e., mutants) against a test suite in order to evaluate the quality of the test suite. In this paper, we first describe a general fault model for concurrent programs and some limitations of previously developed sets of first-order concurrency mutation operators. We then present our new mutation testing approach, which employs synchronization-centric second-order mutation operators that are able to generate subtle concurrency bugs not represented by the first-order mutation. These operators are used in addition to the synchronization-centric first-order mutation operators to form a small set of effective concurrency mutation operators for mutant generation. Our empirical study shows that our set of operators is effective in mutant generation with limited cost and demonstrates that this new approach is easy to implement.

1 Introduction

Mutation testing is a white-box fault-based software testing technique that uses mutants, slightly modified variants of the program source code or byte code, to characterize the effectiveness of a testing suite and locate weaknesses in the test data or program that are seldom or never exposed during normal execution [9]. Mutation testing is based on the *Competent Programmer Hypothesis* and the *Coupling Effect Hypothesis*. The *Competent Programmer Hypothesis* assumes that programmers are competent and normally write programs that are close to perfect; program faults are syntactically small and can be corrected with a few small code modifications [1, 9]. The *Coupling Effect Hypothesis* states that complex bugs in software are closely coupled to small, simple bugs. Thus, mutation testing can be effective in simulating complex real-world bugs [9, 23].

Mutation testing typically involves three stages: (1) Mutant generation, the goal of which is the generation of mutants of the program through inserting bugs. (2) Mutant execution, the execution of test cases against both the original program and the mutants. (3) Result analysis, to check the *mutation score*, *i.e.*, the percentage of nonequivalent mutants that are *killed* by the test suite [26, 23]. A mutant is *equivalent* to the original program if the mutant and the original program always produce the same output, hence no test case can distinguish between the two [8]. A mutant is considered *killed* by the test suite if the execution result of the mutant is different from the result of the original program [25]. A test data set is said to be *adequate* if its mutation score is 100% [8, 24].

For the first stage, a predefined set of mutation operators are used to generate mutants from program source code or byte code. A *mutation operator* is a rule that is applied to a program to create mutants [25]. Mutants containing one simple fault are called *first-order mutants* and mutants containing two simple faults are called *second-order mutants* [26]. Researchers have developed many sets of mutation operators [25, 17], targeting a variety of programming languages. For example, Delamaro *et al.* and Bradbury *et al.* have proposed different set of mutation operators for concurrent Java programs [7, 4]. Our empirical study and analysis shows that some subtle concurrency bugs are not generated by any of these proposed first-order mutation operators. Our study further shows that a large portion of these operators are not effective in mutant generation: the majority of the mutants are generated by a small subset of the mutation operators, generally those that are synchronization-centric, *i.e.*, directly relating to the synchronization of different processes or threads. Based on a general fault model for concurrent programs and our analysis of the limitations in prior work, we present our new mutation testing approach, which employs synchronization-centric second-order mutation operators that are able to generate subtle concurrency bugs not

represented by the first-order mutation. These operators are used in addition to the synchronization-centric first-order mutation operators to form a small set of effective concurrency mutation operators that can be used in mutant generation. Our empirical study shows that our small set of operators is effective in mutant generation with limited cost and demonstrates that this new approach is easy to implement. The initial analysis of the possible implications of our results has potential impact on the Coupling Effect Hypothesis, indicating that possibly the coupling effect is weaker in concurrent programs than in sequential programs.

The remainder of this paper is structured as follows. In Section 2, we describe our fault model for concurrent programs. In Section 3, we present the limitations of some previous work. In Section 4, we present our new approach. In Section 5, we present our empirical study. Lastly, we discuss the related work before we conclude.

2 Fault Model for Concurrent Programs

Testing concurrent programs is difficult. It is generally impossible or impractical to exhaustively test all combinations of input values or cover all possible control or data flow paths in sequential programs but even more so in concurrent programs; nevertheless, test suites can and must be constructed according to various criteria to attempt to find bugs. In order to develop a set of concurrency mutation operators that are able to model subtle concurrency bugs, we employ a general fault model that is based on the concurrency bug patterns and the synchronization mechanisms. Our definition of *fault* is a programming error that leads to an erroneous result in some programs during execution.

2.1 Concurrency Bug Patterns

Some prior research on concurrency bug patterns has been done. [13] and [20] described taxonomy of common concurrency bugs. [12] compiled a benchmark of concurrency bugs. [6] and [21] described some empirical studies on concurrency bugs. We consolidate the common concurrency faults from these prior researches that we consider for mutation operators to model and present them below.

- Data Race: *Data race* condition happens when multiple threads read and write the same data, and the outcome of the execution depends on the particular order in which the accesses happen [6]. It is also called *thread interference*.
- Memory Inconsistency: *Memory inconsistency* errors occur when different threads have inconsistent views of the same variable.
- Atomicity Violation: *Atomicity violation* error is caused by concurrent execution of multiple threads violating the atomicity of a certain code region [21].
- Deadlock: *Deadlock* happens when multiple threads are blocked forever, waiting for each other.
- Livelock: *Livelock* happens when two threads are busy responding to each other and make no progress.
- Starvation: *Starvation* happens when a thread is unable to gain regular access to shared resources and is unable to make progress.
- Suspension: *Suspension* happens when a thread suspends or waits indefinitely.

2.2 Synchronization Mechanism

Concurrent programs rely on synchronization to ensure correct program execution. There are two main synchronization mechanisms: synchronization using shared memory and synchronization using message passing. For programming models that use shared memory synchronization (*e.g.*, Java and C#), the threads communicate primarily by sharing access to fields and the objects reference fields refer to. The synchronization aims to avoid thread interference and memory consistency errors. For programming models that use message passing (*e.g.*, Erlang [3] and Microsoft Asynchronous Agents Library [22]), the concurrent agents or actors in the programs communicate with each other through exchanging messages and use the synchronization to avoid problems in the message communications.

3 Limitations of First-Order Concurrency Mutation Operators

To measure the testability of concurrent Java programs, Ghosh described mutation based on two mutation operators that remove the keyword `synchronized` [14]. Long *et al.* tested mutation-based exploration for concurrent Java components [19]. Delamaro *et al.* proposed a set of 15 concurrency mutation operators for Java [7]. Later, Bradbury *et al.* proposed a new set of 24 concurrency mutation operators for Java [4]. The operators they proposed are all first-order mutation operators. We have investigated these mutation operators in our empirical study and identified some of their limitations.

3.1 Subtle Concurrency Bugs Are Not Generated

The first important limitation we found is that some subtle concurrency bugs are not generated by any of these proposed first-order mutation operators. This limitation could lead to loss of comprehensive representation of common concurrency bugs by the mutants, thus reducing the reliability of the mutation score that follows. We give two examples in the following subsections.

3.1.1 Data Race Example

The following code fragment from the *LinkedList* program, a Java program from the IBM concurrency benchmark programs repository [12], inserts an element to the end of a

list. Another process, not shown here, reads the list. The synchronized method first starts from the top of the list (line 4), then moves to the end of the list via a loop (line 5) before inserts the object `x` to the end (line 6). Suppose we only apply first-order mutation operators (*e.g.*, removing `synchronized` keyword from line 2 or deleting a statement from line 4 to 6), the mutant does not represent a feasible programming error that line 2 is not synchronized and line 5's error causes the node index `itr` does not move to the end of the list properly. The combined error in line 2 and line 5 would potentially cause data race because multiple threads would try to write to the header of the list without synchronization and other threads might read the list at the same time. The outcome of the execution would depend on the thread schedule and which thread made the last call to the method because different threads all try to update the header of the list. By definition, this is a data race condition. To apply either operator independently is not going to create the same fault because under single application of either first-order mutation operator, data race would less likely happen and their mutants would represent different kind of faults.

```
1   /* Inserts element to the end of list */
2   public synchronized void addLast( Object x )
3   {
4      MyListNode itr = this._header;
5      while( itr._next != null ) itr = itr._next;
6      insert(x,new MyLinkedListItr(itr));
7   }
```

3.1.2 Deadlock Example

Incorrect use of synchronization can result in two or more threads waiting for each other to release the locks on the synchronized objects, forming a deadlock circle. As shown in the following example code for money transfer between two accounts, the line 5 and 6 in the original code may be incorrectly programmed in a nested synchronized block, which makes the deadlock possible. For example, two threads with execution of line 9 and 10 simultaneously would lead to deadlock since each thread will be waiting in a circle for the other thread to release required lock. This kind of deadlocks that require changes in more than one place are not generated by any first-order operator.

```
1   void TransferMoney(Acct a, Acct b, int amount) {
2      synchronized(a) {
3         a.debit(amount);
4      }
5       synchronized(b) {
6         b.credit(amount);
7      }
8   }
1   void TransferMoney(Acct a, Acct b, int amount) {
2       synchronized(a) {
3         synchronized(b) {  //first change
4            a.debit(amount);
5            b.credit(amount);  //second change
6         }
7      }
8   }
9   Thread1.run() { TransferMoney(a,b,10); }
10  Thread2.run() { TransferMoney(b,a,20); }
```

3.2 A Large Portion of Mutation Operators Do Not Generate Any Mutant

Our empirical study shows that a large portion of existing mutation operators are not effective in generating mutants. For example, several previously proposed mutation operators for concurrent Java, including MSF, MXC, MBR, RCXC, ELPA, EAN, RSTK, RFU, RXO and EELO [7, 4], did not generate any mutants in our experiments. Some others, including MXT, RNA, RJS, InsNegArg, and ReplTargObj [7, 4], generated very few mutants. In our assessment, over half of the total number of operators are non-performing mutation operators, *i.e.*, operators that do not generate any new mutant. Most of the performing ones are related to mutation of a synchronized method or block.

4 Approach

4.1 Synchronization-Centric Second-Order Concurrency Mutation Operators

Our new mutation testing approach is based on our fault model and our analysis of the limitations of some previous work. We use synchronization-centric second-order concurrency mutation operators to construct subtle concurrency bugs that are not represented by the first-order mutation. While, a random and brute-force approach without any reduction would lead to $n * n$ second-order mutation operators based on n first-order mutation operators. To reduce the number of second-order mutation operators and mutant execution cost, we employ two steps of reduction. We first choose one of the two first-order mutation operators to be a synchronized method or block related modification and the other first-order operator to perform code changes related to the same synchronized method or block. For example, in concurrent Java, there are five first-order mutation operators related to synchronized methods [7, 4], we choose two out of the five in the same category. Then we evaluate the chosen two first-order mutation operators to see if their combination can generate mutants that resemble some possible faults due to programming mistakes and only keep those meaningful combinations. This second reduction step through selection based on domain knowledge further reduces the amount of second-order mutation operators and leads to fewer unnecessary or redundant mutants.

After the set of synchronization-centric second-order concurrency mutation operators are chosen, they are combined with the synchronization-centric first-order mutation operators to form a smaller set of mutation operators for mutant generation.

4.2 Example Mutation Operators for Java

Table 1 lists the synchronization-centric second-order concurrency mutation operators for Java, as an example of synchronization using shared memory. We describe each operator with example code in the following subsections.

Table 1. Second-Order Concurrency Mutation Operators for Java

sync method	RKSN+RSSN	Remove `synchronized` Keyword and a Statement from Synchronized Method
	AKST+MASN	Add `static` Keyword and Modify Argument with Constant to Synchronized Method
	RKSN+MASN	Remove `synchronized` Keyword and Modify Argument with Constant
sync block	RSNB+RSSB	Remove `synchronized` Block and a Statement from Synchronized Block
	MOSB+RSSB	Modify `synchronized` Object and Remove a Statement from Synchronized Block
	MOSB+MVSB	Modify `synchronized` Object and Move Statement(s) Out of Synchronized Block

4.2.1 RKSN+RSSN

The RKSN+RSSN operator removes a `synchronized` keyword and a statement from a synchronized method. This operator simulates programming errors that can potentially lead to data race, memory inconsistency, and deadlock. The data race described in Section 3.1.1 can be constructed by this operator.

```
/* Original Code */
public synchronized void proc(Object A) {
   <statement 1>
   <statement 2>
}
/* RKSN+RSSN Mutant 1 */
public void proc(Object A) {  // sync removed
   ...// statement removed
   <statement 2>
}
/* RKSN+RSSN Mutant 2 */
public void proc(Object A) {  // sync removed
   <statement 1>
   ...// statement removed
}
```

4.2.2 AKST+MASN

The AKST+MASN operator adds a `static` keyword and modifies an argument with a constant to a synchronized method. This operator simulates programming errors that can potentially lead to data race and memory inconsistency.

```
/* Original Code */
public synchronized void send(String m) {...}
/* AKST+MASN Mutant */
public static synchronized void send(String n) {...}
```

4.2.3 RKSN+MASN

The RKSN+MASN operator removes a `synchronized` keyword and modifies an argument with a constant to a synchronized method. This operator simulates programming errors that can potentially lead to data race and memory inconsistency.

```
/* Original Code */
public synchronized void send(String m) {...}
/* AKST+MASN Mutant */
public void send(String n) {...}
```

4.2.4 RSNB+RSSB

The RSNB+RSSB operator removes the synchronized block and a statement from a synchronized block. This operator simulates programming errors that can potentially lead to data race, memory inconsistency, and deadlock.

```
/* Original Code */
synchronized (this) {
   <statement 1>
   <statement 2>
}
/* RSNB+RSSB Mutant 1 */
... // removed
    ... // removed
   <statement 2>
...
/* RSNB+RSSB Mutant 2 */
... // removed
   <statement 1>
    ... // removed
...
```

4.2.5 MOSB+RSSB

The MOSB+RSSB operator modifies a synchronized object and removes a statement from a synchronized block. This operator simulates programming errors that can potentially lead to data race and memory inconsistency.

```
/* Original Code */
synchronized(obj1) {
   <statement 1>
   <statement 2>
}
/* MOSB+RSSB Mutant */ // object modified
synchronized(newobj) {
   <statement 1>
    ... // removed
}
/* MOSB+RSSB Mutant */
synchronized(newobj) { // object modified
    ... // removed
   <statement 2>
}
```

4.2.6 MOSB+MVSB

The MOSB+MVSB operator modifies a synchronized object and moves statement(s) out of a synchronized block. This operator simulates programming errors that can potentially lead to data race, deadlock, memory inconsistency, and atomicity violation. The deadlock described in Section 3.1.2 can be constructed by this operator, *i.e.*, moving two lines of code including the synchronized block.

```
/* Original Code */
synchronized(obj1) {
   <statement 1>
   <statement 2>
    ...
}
/* MOSB+MVSB Mutant 1*/
<statement 1>  // moved
synchronized(newobj) { // object modified
   <statement 2>
    ...
}
/* MOSB+MVSB Mutant 2*/
<statement 1>  // moved
<statement 2>  // moved
synchronized(newobj) { // object modified
    ...
}
```

4.3 Example Mutation Operators for Erlang

Table 2 lists the synchronization-centric second-order concurrency mutation operators for Erlang, as an example of synchronization using message passing. The CRT (*i.e.*,

Change Reference Type) refers to changing a message reference from *Send by Ref* to *Send by Val*, and vice versa [15]. The CST (*i.e.*, Change Synchronization Type) refers to changing a message's synchronization method from *Sync Send* to *Async Send*, and vice versa. Since these mutation operators are self-explanatory, we do not give detailed example code here.

Table 2. Second-Order Concurrency Mutation Operators for Erlang

Messaging	CRT+MMP	Change reference type and modify message parameter
	CRT+RMP	Change reference type and reorder message parameter
	CRT+MMN	Change reference type and modify message name
	CRT+MMR	Change reference type and modify message recipient
	CST+MMP	Change sync type and modify message parameter
	CST+RMP	Change sync type and reorder message parameter
	CST+MMN	Change syn type and modify message name
	CST+MMR	Change syn type and modify message recipient
Constraint	CRT+RC	Change reference type and remove constraint
	CRT+MC	Change reference type and modify constraint
	CST+RC	Change syn type and remove constraint
	CST+MC	Change syn type and modify constraint

5 Empirical Study

5.1 Implementation

We developed an Eclipse Plug-in [11] named BUGGEN that is able to automate mutant generation after the specific mutation operator is selected. Eclipse is a popular integrated development environment (IDE) with an extensible plug-in system. Building BUGGEN as an Eclipse Plug-in leverages the functionalities of the Eclipse and simplifies software development. During our implementation, we found our set of mutation operators is easy to implement. In our empirical study, we focus on concurrent Java.

5.2 Example Programs

We use the following four example programs in our experiments to study mutant generation, as well as the cost and effectiveness of each proposed operator,

- *Webserver*, a Java web server program that supports concurrent client connections and synchronization [2].
- *Chat*, a Java chat program that supports multiple clients exchanging messages [10].
- *Miasma*, a graphical Java applet program from the NIH web-site [18]. It supports synchronization and uses `wait(t)` for prior pixels to be accepted before triggering another one.
- *LinkedList*, a modified Java program from the IBM concurrency benchmark programs repository [12]. The original program was developed to emulate the concurrency bug in using Java linked list, which is a non-synchronized collection.

We select the above example programs because they all employ different concurrency features and these programs are diversified in terms of type, size, coding style, applied field, and developer. These programs are representative in demonstrating common programming practices using concurrent Java. Table 3 lists some statistical information for each program's source code.

Table 3. Example programs

Program Name	LOC	classes	sync methods	sync blocks
Webserver	125	6	11	2
Chat	482	4	10	2
Miasma	360	1	0	2
LinkedList	421	5	1	1
Total	1,388	16	22	7

5.3 Mutant Generation Results and Analysis

In our experiments, we apply each of the mutation operators listed in Table 1, along with the synchronization-centric first-order mutation operators, on the example programs, count the number of mutants generated by each operator for each program, and then examine these mutants. From our experiments, we found that over half of the first-order mutation operators, especially those that are not related to synchronization, are not effective in generating mutants. Synchronization-centric mutation operators generate the majority of the mutants. Our quantitative data and summations for each category are recorded in the histogram chart presented in Figure 1. Details for each operator and the example programs can be found in our technical report [27]. The vertical axis shows the number of mutants. Most synchronization-centric mutation operators, in particular the second-order ones, are effective in mutant generation.

Our empirical study demonstrates that the second-order mutation operators generate subtle concurrency bugs not represented by the first-order mutation; our mutant generation effort is limited; fewer percentages of equivalent mutants are generated. Second-order operators tend to decrease the percentage of equivalent mutants [26].

6 Related Work

Some prior studies have been done on mutation testing for concurrent programs [17]. Carver described deterministic execution mutation testing and debugging of concurrent programs using synchronization-sequence [5]. Researchers have developed many sets of mutation operators [25, 17], targeting a variety of programming languages. Other than the mutation operators for concurrent Java, Jagannath *et al.* have proposed a set of mutation operators for actor programming model [15]. Our synchronization-centric second-order mutation operators for message passing also apply to the actor programming model.

For higher-order mutation, Polo *et al.* studied mutation cost reduction using second-order mutants [26]. Jia *et al.* described some general cases of higher-order mutation and related algorithms [16]. In our approach, we used second-order mutation to construct some subtle concurrency faults.

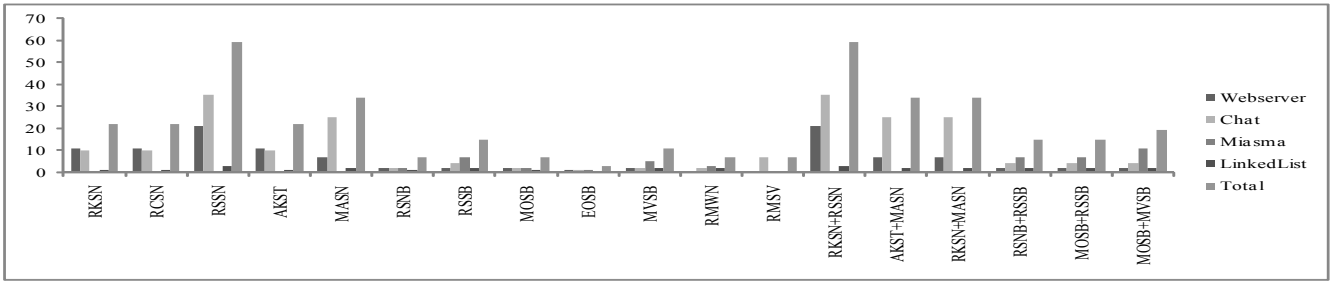

Figure 1. Number of mutants generated per operator

By keeping a small number of second-order mutation operators based on synchronization and reduction through domain analysis, we avoided the drastic growth of the number of mutants, thus avoiding higher computing cost in mutant execution. To the best of our knowledge, our work is the first study of the second-order mutation operators specifically for concurrent programs.

7 Conclusion

This paper first described a general fault model for concurrent programs and some limitations of previously developed sets of first-order concurrency mutation operators. We then presented our new mutation testing approach, which employs synchronization-centric second-order mutation operators that are able to generate subtle concurrency bugs not represented by the first-order mutation. These operators are used in addition to the synchronization-centric first-order mutation operators to form a small set of effective concurrency mutation operators that can be used in mutant generation. We developed an Eclipse Plug-in named BUGGEN to automate the mutant generation using these operators. Our empirical study showed that our set of mutation operators is effective in mutant generation with limited cost and this new approach is easy to implement. For future work, we plan to evaluate some concurrency testing suites using the set of mutation operators.

8 Acknowledgments

Wu and Kaiser are members of the Programming Systems Laboratory, funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 2 U54 CA121852-06.

References

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, Georgia, 1979.

[2] J. Aldrich, E. G. Sirer, C. Chambers, and S. J. Eggers. Comprehensive synchronization elimination for java. *Science of Computer Programming*, 47(2-3):91–120, 2003.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993, Second Edition.

[4] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrent java (j2se 5.0). In *Proceedings of the Second Workshop on Mutation Analysis (Mutation '06)*, pages 11–11. IEEE Computer Society, 2006.

[5] R. Carver. Mutation-based testing of concurrent programs. In *Proceedings of the International Test Conference*, pages 845–853, 1993.

[6] S.-E. Choi and E. C. Lewis. A study of common pitfalls in simple multi-threaded programs. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*. ACM, 2000.

[7] M. Delamaro, M. Pezzé, A. M. R. Vincenzi, and J. C. Maldonado. Mutant operators for testing concurrent java programs. In *XV Simpósio Brasileiro de Engenharia de Software*, pages 272 – 285, Rio de Janeiro, RJ, Brasil, 2001.

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

[9] 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, 1978.

[10] D. J. Eck. Chat. available at `http://math.hws.edu/eck/cs124/s06/lab11/index.html`, 2006.

[11] Eclipse. Eclipse.org. available at `http://www.eclipse.org`, 2010.

[12] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, page 266, 2004.

[13] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 2003.

[14] S. 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*, pages 17–25, 2002.kim

[15] V. Jagannath, M. Gligoric, S. Lauterburg, D. Marinov, and G. Agha. Mutation Operators for Actor Systems In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pp. 157–162, 2010.

[16] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258, 2008.

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions of Software Engineering*, 2010.

[18] JRP. Miasma. available at `http://rsb.info.nih.gov/miasma/Miasma.java`, 2010.

[19] B. Long, R. Duke, D. Goldson, P. Strooper, and L. Wildman. Mutation-based exploration of a method for verifying concurrent java components. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, page 265, 2004.

[20] B. Long and P. Strooper. A classification of concurrency failures in java components. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*, pp. 8, 2003.

[21] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*. ACM, 2008.

[22] Microsoft Asynchronous Agents Library http://msdn.microsoft.com/en-us/library/dd492627(VS.100).aspx

[23] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, 1992.

[24] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[25] A. J. Offutt and R. H. Untch. Mutation 2000: uniting the orthogonal. *Mutation Testing for the New Century*, pages 34–44, 2001.

[26] M. Polo, M. Piattini, and I. García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19(2):111–131, 2009.

[27] L. Wu and G. Kaiser. Empirical study of concurrency mutation operators for java. Technical Report CUCS-041-10, Department of Computer Science, Columbia University, 2010.