The *ConMAn* Operators

User Guide

[DRAFT]

Jeremy S. Bradbury
Faculty of Science (Computer Science)
University of Ontario Institute of Technology (UOIT)
Oshawa, Ontario, Canada
jeremy.bradbury@uoit.ca
`http://faculty.uoit.ca/bradbury/`

Last Modified: August 2007

Contents

1 What is *ConMAn*?

The *ConMAn* (*Con*currency *M*utation *An*alysis) operators were developed for mutating concurrent source code written in Java (J2SE 5.0)[1]. There are 24 *ConMAn* operators (see Table **??**) and each operator was based on real concurrency bug patterns. Each operator is implemented in TXL - a source transformation language (`http://www.txl.ca`).

The *ConMAn* operators can be 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 *ConMAn* has been used previously as a comparative metric for different fault detection tools 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. Furthermore, the *ConMAn* operators should be viewed as a complement not a replacement

[1]You can use the operators with previous versions of Java (1.4 and earlier)

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 1: The *ConMAn* operators for Java

for the existing operators used in tools like MuJava. For example, using the *ConMAn* operators can direct mutate concurrency mechanisms like synchronization while using the method level operators can mutate other parts of the source code that may indirectly affect concurrency mechanisms.

2 Downloading and Installing *ConMAn*

2.1 Downloading *ConMAn*

The *ConMAn* Operators are free for download and are distributed under the <**Insert license name here**>. There are two options for downloading *ConMAn*:

1. Download executables. The first option will download an archive file (conman1.0_exe.zip) containing an executable program for each of the 24 *ConMAn* operators, a script to run all of the operators in batch mode and an example Java program to mutate.

2. Download TXL source files. The second option will download an archive file (conman1.0_src.zip) containing the TXL source files for each operator, the batch mode script and the example program. If you download the TXL source files you will have to obtain the TXL compiler/interpreter from `http://www.txl.ca`.

We recommend that most users download the *ConMAn* executables. The source file option is only necessary if you wish to modify the operators. We are currently developing an automatic system for downloading *ConMAn* which will be available at `http://faculty.uoit.ca/bradbury/conman/download.html` by Fall 2007.

If you are interested in obtaining *ConMAn* before that please e-mail `jeremy.bradbury@uoit.ca`.

2.2 Installing *ConMAn*

...

3 Running *ConMAn*

3.1 Running *ConMAn* from the command-line

Each *ConMAn* Operator can be run separately using the following command:

```
txl <infilename> <operator>.Txl - -outfile <outfilename> -outdir <outputpath>
```

Additionally the operators also include a script called ConMAn.sh which can be used to execute all of the operators as follows:

...

In the future we plan to develop a GUI interface for *ConMAn*.

3.2 Running *ConMAn* within the *ExMAn* Framework

ConMAn was originally developed with the intention of being used as a plugin to the *ExMAn* Framework. For instructions on using the *ConMAn* operators with *ExMAn* please see the *ExMAn* website `http://www.uoit.ca/bradbury/exman/`.

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

4 The *ConMAn* Operators

ConMAn includes 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 **??** – which demonstrates that the operators provide coverage over the J2SE 5.0 concurrency mechanisms.

A complete list of the operators we will be presenting in this section was provided in Table **??**. 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 [**?**] 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 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 [**?**]. 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.

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.

4.1.1 MXT - Modify Method-X Timeout

The MXT operator can be applied to the wait(), sleep(), and join() method calls (see Appendix **??**) 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.

4.1.2 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.

4.1.3 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.

4.1.4 MSF - Modify Semaphore Fairness

A a semaphore maintains a set of permits for accessing a resource (see Appendix **??**). 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 );
...
```

4.1.5 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 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:

[3] `java.util.concurrent` documentation

Original Code:

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

M*X*C 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(){
        }
    });
...
```

M*X*C 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.

4.1.6 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.

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).

4.2.1 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 ...
```

RT*X*C Mutant for wait():

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

4.2.2 RC*X*C - Remove Concurrency Mechanism Method-X Call

The RC*X*C 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 RC*X*C operator in a ReentrantLock or a ReentrantReadWriteLock with a call to the unlock() method. The RC*X*C 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();
}
...
```

RC*X*C Mutant for a Lock:

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

The RC*X*C 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();
...
```

RC*X*C Mutant for a Semaphore:

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

Another RC*X*C 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 RC*X*C operator for other concurrency mechanisms we will not provide any additional examples.

4.2.3 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();...
```

4.2.4 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);...
```

4.2.5 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.

4.2.6 EAN - Exchange 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);
...
```

EAN Mutant:

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

4.3 Modify Keywords: Add and Remove

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

4.3.1 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()
{ ... }
```

4.3.2 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()
{ ... }
```

4.3.3 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()
{ ... }
```

4.3.4 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()
{ ... }
```

4.3.5 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>
    ...
```

4.3.6 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;
```

4.3.7 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();
...
```

4.4 Switch Concurrent Objects

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

4.4.1 R*X*O - 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();
...
```

R*X*O 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 R*X*O operator when 2 or more objects exist of type Semaphore, CountDownLatch, CyclicBarrier, Exchanger, and more. For example consider the application of the R*X*O 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();
...
```

R*X*O 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();
...
```

R*X*O 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();
...
```

4.4.2 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 **??**). 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();
}
...
```

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.

4.5.1 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>
...
```

4.5.2 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.

4.5.3 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>
...
```

4.5.4 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>
...
```

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 **??** 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

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 3: Concurrency bug patterns vs. *ConMAn* mutation operators

do not have mutation operators are typically more specific complex patterns and the development of general operators related to these patterns is not feasible.

5 The *ConMAn* Team

ConMAn was developed by:

- Jeremy S. Bradbury, Assistant Professor, Faculty of Science (Computer Science), University of Ontario Institute of Technology
- James R. Cordy, Professor, School of Computing, Queen’s University
- Juergen Dingel, Associate Professor, School of Computing, Queen’s University

ConMAn is currently maintained by Jeremy Bradbury and all comments or bug reports should be sent to `jeremy.bradbury@uoit.ca`.

References

[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.

[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.

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

[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.

A Java Concurrency Mechanisms

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 [**?**]. 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 [**?**].

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.

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.

Other Concurrency Mechanisms. In J2SE 5.0, additional mechanisms to support concurrency were added as part of the `java.util.concurrent` library[2]:

- *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(), awaitUniterruptibly(), 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.

[2] `java.lang.Thread` documentation

[2] 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().