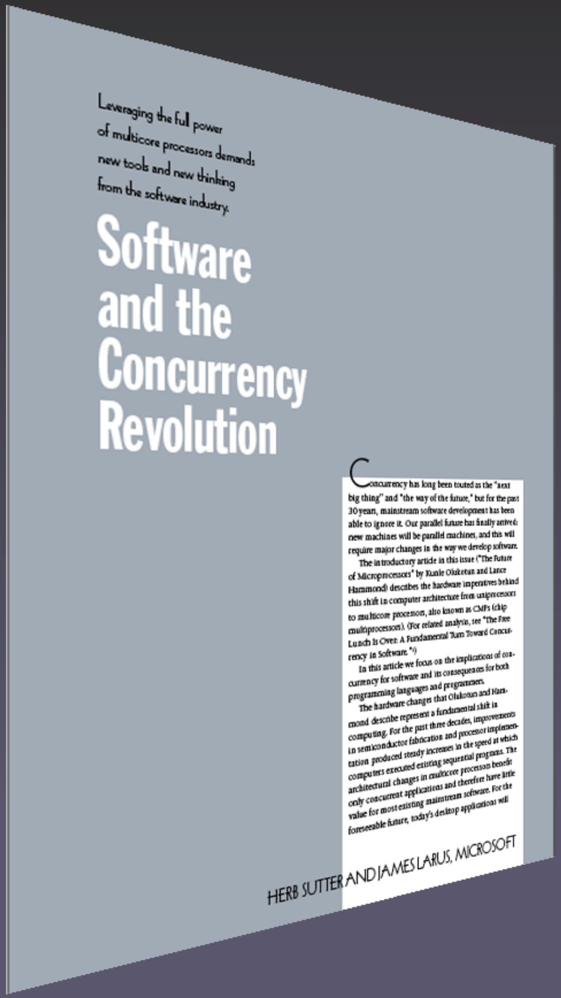
Defining a Catalog of Programming Anti-Patterns for Concurrent Java

Jeremy S. Bradbury, Kevin Jalbert
Software Quality Research Group
Faculty of Science (Computer Science)
University of Ontario Institute of Technology
Oshawa • Ontario • Canada
jeremy.bradbury@uoit.ca, kevin.jalbert@mycampus.uoit.ca

SPAQu'09 • October 25, 2009

SUPPORTED BY



Leveraging the full power of multicore processors demands new tools and new thinking from the software industry.

Software and the Concurrency Revolution

Concurrency has long been touted as the "next big thing" and "the way of the future," but for the past 30 years, mainstream software development has been able to ignore it. Our parallel future has finally arrived: new machines will be parallel machines, and this will require major changes in the way we develop software.

The introductory article in this issue ("The Future of Microprocessors" by Kunle Olukotun and Lance Hammond) describes the hardware imperatives behind this shift in computer architecture from uniprocessors to multicore processors, also known as CMPs (chip multiprocessors). (For related analysis, see "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software."¹)

In this article we focus on the implications of concurrency for software and its consequences for both programming languages and programmers.

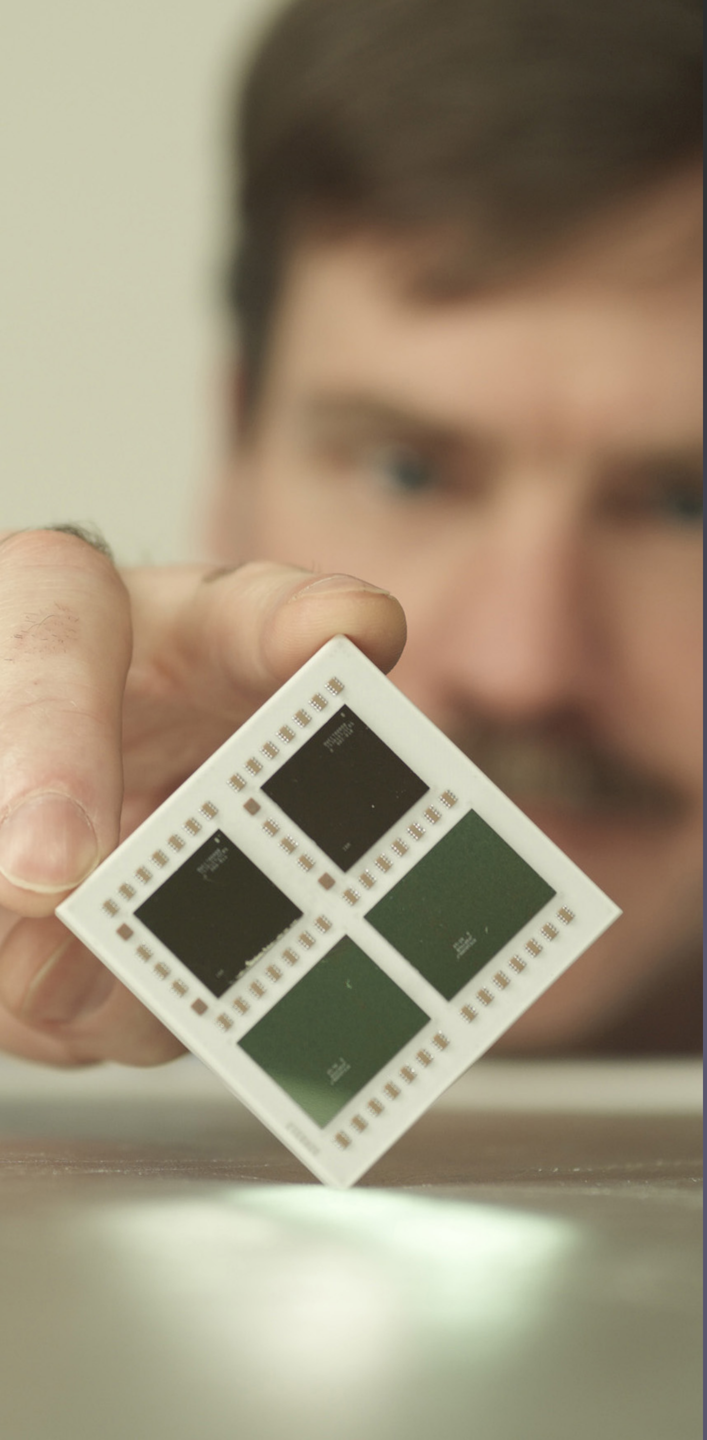
The hardware changes that Olukotun and Hammond describe represent a fundamental shift in computing. For the past three decades, improvements in semiconductor fabrication and processor implementation produced steady increases in the speed at which computers executed existing sequential programs. The architectural changes in multicore processors benefit only concurrent applications and therefore have little value for most existing mainstream software. For the foreseeable future, today's desktop applications will

HERB SUTTER AND JAMES LARUS, MICROSOFT

“...humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings...”

- Herb Sutter & James Larus, Microsoft [SL05]

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



In the future applications will need to be **concurrent** to fully exploit CPU throughput gains [Sut05]

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

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

- Edward A. Lee [Lee06]

COVER FEATURE

The Problem with Threads

Edward A. Lee
University of California, Berkeley

For concurrent programming to become mainstream, we must discard threads as a programming model. Nondeterminism should be judiciously and carefully introduced where needed, and it should be explicit in programs.

Concurrent programming is difficult,[1] yet many technologists predict the end of Moore's law will be answered with increasingly parallel computer architectures—multicore or chip multiprocessors (CMPs).[2] If we hope to achieve continued performance gains, programs must be able to exploit this parallelism.

Automatic exploitation of parallelism in sequential programs, through either computer architecture techniques such as dynamic dispatch or automatic parallelization of sequential programs,[3] offers one possible technical solution. However, many researchers agree that these automatic techniques have been pushed to their limits and can exploit only modest parallelism. Thus, programs themselves must become more concurrent.

Understanding why concurrent programming is so difficult can help us solve the problem. The physical world is highly concurrent, and our very survival depends on our ability to reason about concurrent physical dynamics. This reasoning doesn't extend to concurrent programs because we have chosen abstractions that do not even vaguely resemble the physical world's concurrency. We have become so used to these computational abstractions that we have forgotten they are not immutable. The difficulty of concurrent programming is a consequence of these abstractions, and if we can let go of them, the problem will be fixable.

THREADS

In general-purpose software engineering practice, we have reached a point where one approach to concurrent programming dominates all others—namely, threads, sequential processes that share memory. They represent a key concurrency model supported by modern computers, programming languages, and operating systems. Many general-purpose parallel architectures in use today—such as symmetric multiprocessors—are direct hardware realizations of the thread abstraction.

Some applications can use threads very effectively—for example, so-called embarrassingly parallel applications that essentially spawn multiple independent processes such as build tools (PVM gmake) or Web servers. Given these applications' independence, programming is relatively easy and the abstraction being used is more like processes than threads. Where such applications do share data, they do so through database abstractions, which manage concurrency through such mechanisms as transactions. However, client-side applications are not so simple.

Threads are not the only possibility for concurrent programming. In scientific computing, where performance requirements have long demanded concurrent programming, data-parallel language extensions and message-passing libraries—such as PVM, MPI, and OpenMP—dominate over threads for concurrent programming. Computer architectures intended for scientific computing often differ significantly from so-called general-purpose architectures. They commonly support vectors and streams in hardware, for example. However, even in this domain, concurrent programs remain tedious to write. C and Fortran dominate, despite a long history of much better data-parallel languages.

[illegible]

Published by the IEEE Computer Society

May 2006 33

Java Concurrency

- Java concurrency is built around the notion of **multi-threaded** programs
  - sleep(), yield(), join() can affect the status of a thread
- Access to shared data supported primarily through **synchronized** methods and blocks
- Synchronization blocks can be used in combination with **implicit monitor locks**
  - Monitor methods: wait(), notify(), notifyAll()

Java Concurrency

- Explicit Lock: same semantics as the implicit monitor locks plus additional functionality such as timeouts during lock acquisition.
- Semaphore: Maintains a set of permits that restrict the number of threads accessing a resource.
- Latch: Allows threads to wait until other threads complete a set of operations.
- Barrier: A point at which threads from a set wait until all other threads reach that point.
- Exchanger: Allows two threads to exchange objects at a given synchronization point.

```java
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

/**
   * Created by IntelliJ IDEA.
   * User: amit rotstein I.D: 037698867
   * Date: Oct 17, 2003
   * Time: 1:02:13 PM
   * To change this template use Options | File Templates.
   * Modified by J.S. Bradbury (Feb. 2007)
   */
public class Bug implements Runnable {
    static int Num_Of_Seats_Sold = 0;
    int Maximum_Capacity, Num_of_tickets_issued;
    boolean StopSales = false;
    Thread threadArr [];

    public Bug (int size, int cushion) {
        Num_of_tickets_issued = size;
        Maximum_Capacity = Num_of_tickets_issued - cushion;
        threadArr = new Thread [Num_of_tickets_issued];
        //starting the selling of the tickets:
        for (int i = 0;
        i < Num_of_tickets_issued; i ++) {
            System.out.println ("Creating seller thread # " + i);
            threadArr [i] = new Thread (this);
            threadArr [i].start ();
            // "make the sale !!!"
        }
    }

    /**
       * the selling post:
       * making the sale & checking if limit was reached ( and updating "StopSales" ),
       */
    public void run () {
        if (StopSales == false) {
            System.out.println ("Ticket sold");
            Num_Of_Seats_Sold ++;
        }
        synchronized (this) {
            if (Num_Of_Seats_Sold == Maximum_Capacity) {
                System.out.println ("Maximum capacity reach - no ticket  sold");
                StopSales = true;
                // updating
            }
            if (Num_Of_Seats_Sold > Maximum_Capacity) throw new RuntimeException ("bug
found - oversold seats!!");

        }}

}
```

```
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

/**
   * Created by IntelliJ IDEA.
   * User: amit rotstein I.D: 037698867
   * Date: Oct 17, 2003
   * Time: 1:02:13 PM
   * To change this template use Options | File Templates.
   * Modified by J.S. Bradbury (Feb. 2007)
   */
public class Bug implements Runnable {
    static int Num_Of_Seats_Sold = 0;
    int Maximum_Capacity, Num_of_tickets_issued;
    boolean StopSales = false;
    Thread threadArr [];

    public Bug (int size, int cushion) {
        Num_of_tickets_issued = size;
        Maximum_Capacity = Num_of_tickets_issued - cushion;
        threadArr = new Thread [Num_of_tickets_issued];
        //st
        for
        i <


        }
    }

    /**
       * the
       * mak
       */
    public v
        if (


        }
        sync




found - over

        }}

}
```

```
/**
     * the selling post:
     * making the sale & checking if limit was reached ( and updating "StopSales" ),
     */
   public void run () {
       if (StopSales == false) {
           System.out.println ("Ticket sold");
           Num_Of_Seats_Sold ++;
       }
       synchronized (this) {
           if (Num_Of_Seats_Sold == Maximum_Capacity) {
               System.out.println ("Maximum capacity reach - no ticket  sold");
               StopSales = true;
               // updating
           }
           if (Num_Of_Seats_Sold > Maximum_Capacity) throw new RuntimeException ("bug
found - oversold seats!!");

       }}
```

A pattern is defined as something that "...describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [GHJV95].

An anti-pattern defines a recurring *bad* design solution [Mey06].

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, ser. Addison-Wesley Professional Computing Series. Addison Wesley, 1995.

[Mey06] M. Meyer, "Pattern-based reengineering of software systems," in 13th Working Conference on Reverse Engineering (WCRE'06), 2006, pp. 305–306.

Related Work

- Prior to J2SE 5.0, Farchi, Nir, and Ur developed a bug
- pattern taxonomy for Java concurrency [FNU03].
  - The bug patterns are based on common mistakes programmers make when developing concurrent code in practice.
- This taxonomy has been extended to included the `java.util.concurrent` library [BCD06].

[FNU03] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in Proc. of the 1st Int. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003), Apr. 2003.

[BCD06] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in Proc. of the 2nd Workshop on Mutation Analysis (Mutation 2006), Nov. 2006, pp. 83–92.

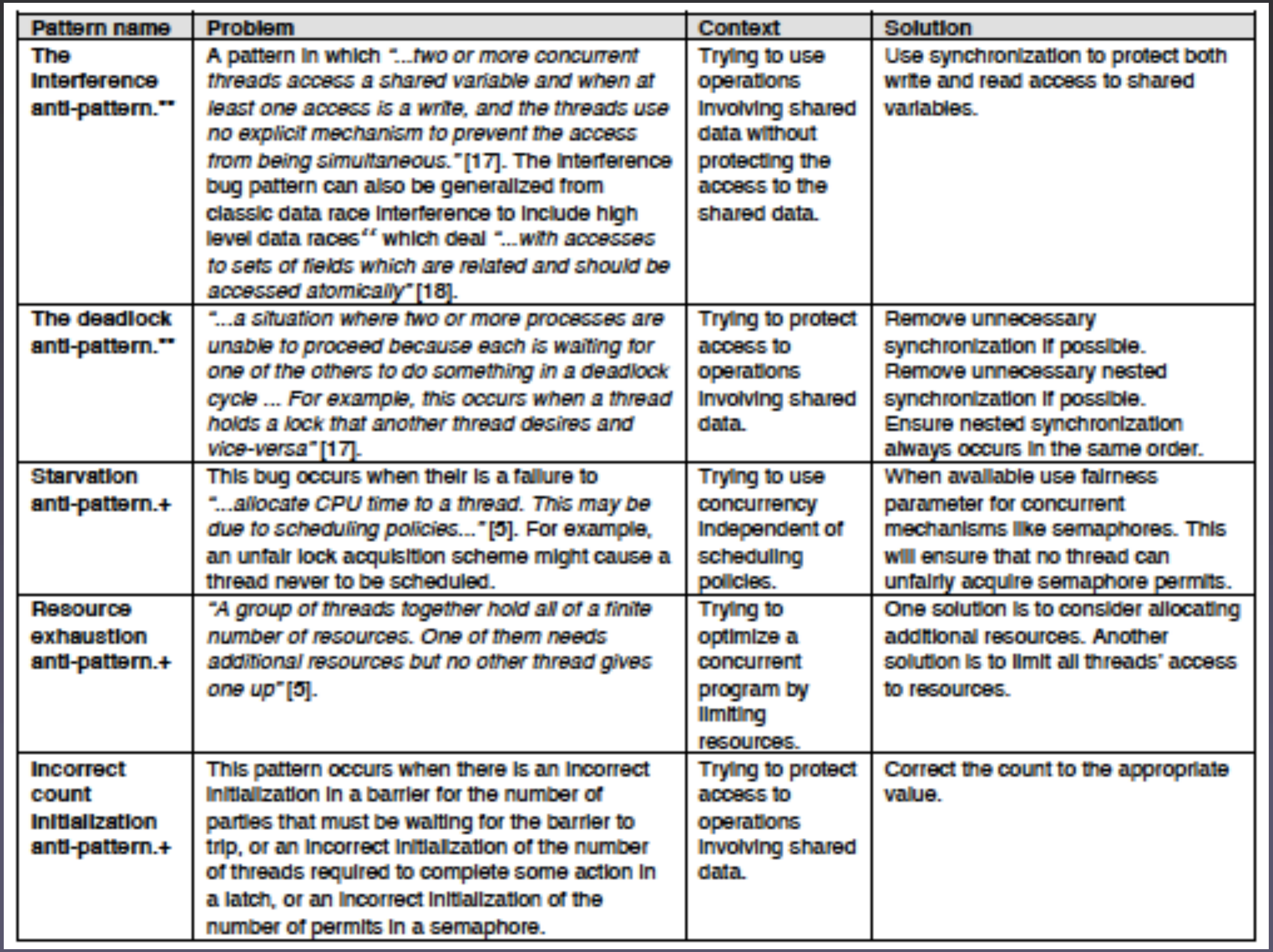
Related Work

- An anti-pattern catalog for Java multithreaded software has already been developed [HAT+04]
  - primarily contains design anti-patterns related to efficiency, quality and style…
- Our work focuses on the identification of anti-patterns based on bugs and includes anti-patterns related to the correctness of the program.
  - Therefore, we believe that both catalogs are complementary

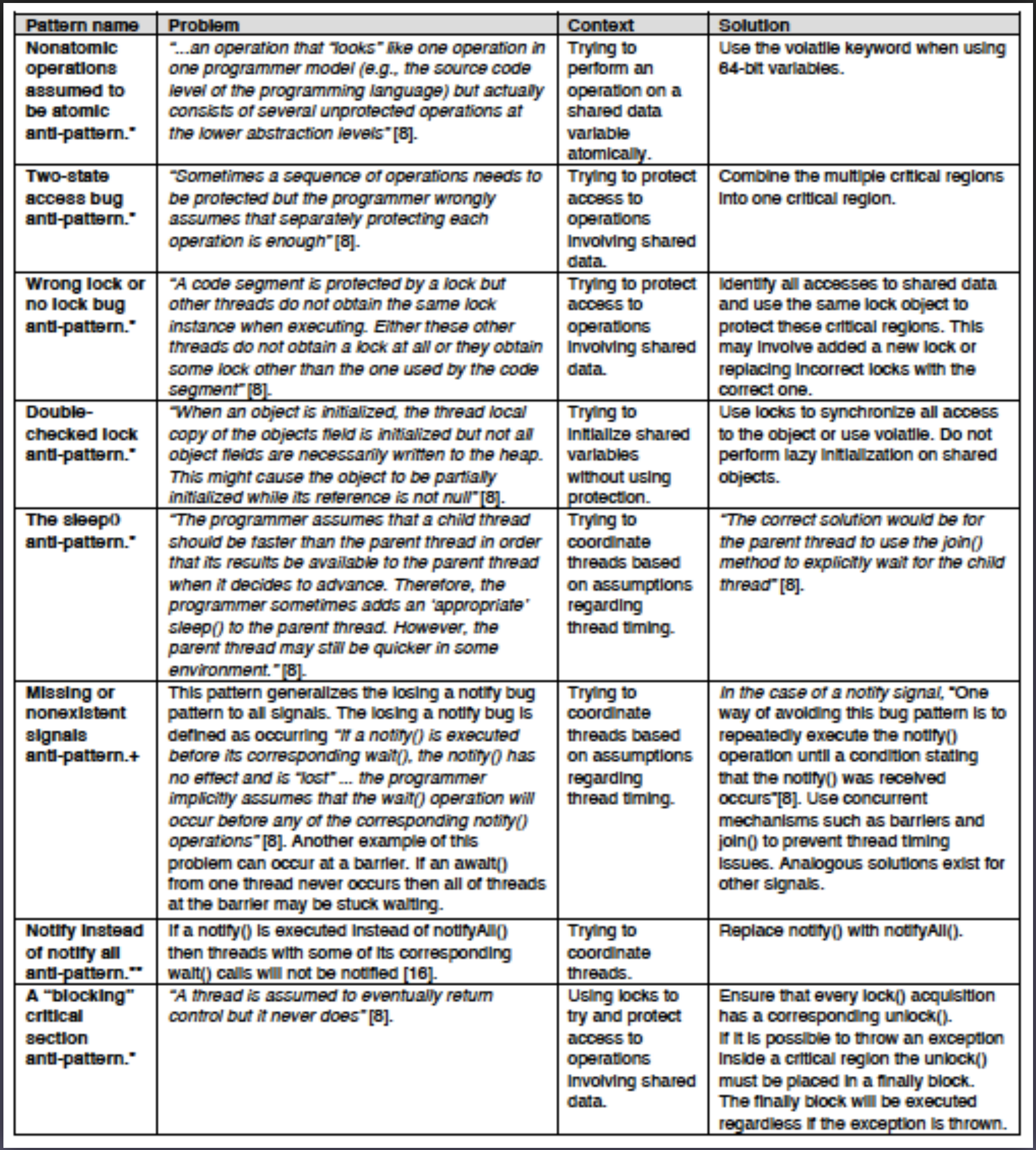
[HAT+04] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, and A. Petrenko, "Antipattern-based detection of deficiencies in Java multithreaded software," in 4th International Conference on Quality Software (QSIC 2004), 2004, pp. 258–267.

Defining Concurrency Anti-Patterns

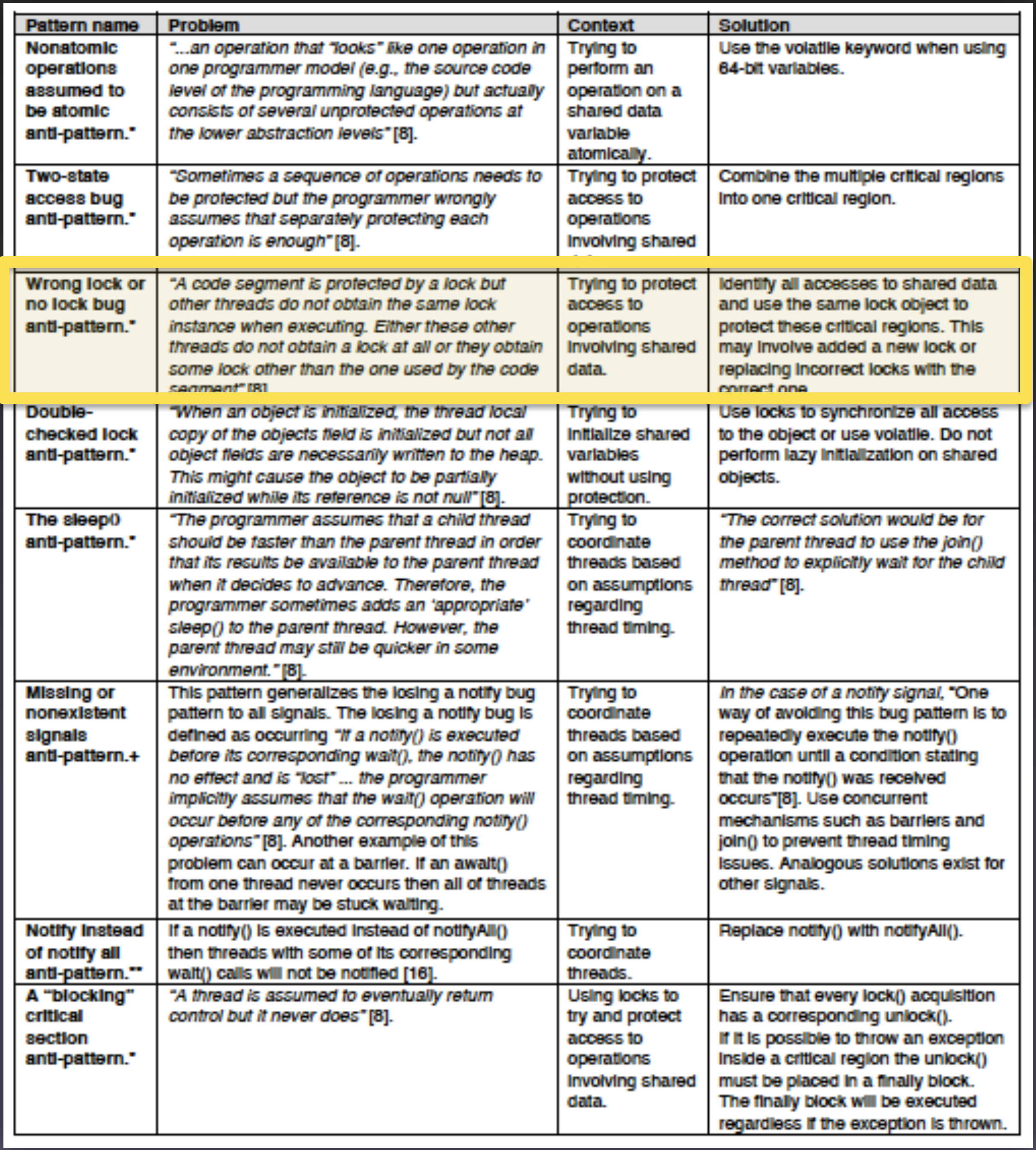
1. pattern name: the anti-pattern name is based on the corresponding bug's name.
2. problem: the problem describes the corresponding bug that is being addressed.
3. context: the context in which the problem often occurs.
4. solution: the solution describes general steps that can be taken to correct the anti-pattern.



Pattern name	Problem	Context	Solution
The interference anti-pattern.**	A pattern in which *"...two or more concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the access from being simultaneous."* [17]. The interference bug pattern can also be generalized from classic data race interference to include high level data races" which deal *"...with accesses to sets of fields which are related and should be accessed atomically"* [18].	Trying to use operations involving shared data without protecting the access to the shared data.	Use synchronization to protect both write and read access to shared variables.
The deadlock anti-pattern.**	*"...a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something in a deadlock cycle ... For example, this occurs when a thread holds a lock that another thread desires and vice-versa"* [17].	Trying to protect access to operations involving shared data.	Remove unnecessary synchronization if possible. Remove unnecessary nested synchronization if possible. Ensure nested synchronization always occurs in the same order.
Starvation anti-pattern.+	This bug occurs when their is a failure to *"...allocate CPU time to a thread. This may be due to scheduling policies..."* [5]. For example, an unfair lock acquisition scheme might cause a thread never to be scheduled.	Trying to use concurrency independent of scheduling policies.	When available use fairness parameter for concurrent mechanisms like semaphores. This will ensure that no thread can unfairly acquire semaphore permits.
Resource exhaustion anti-pattern.+	*"A group of threads together hold all of a finite number of resources. One of them needs additional resources but no other thread gives one up"* [5].	Trying to optimize a concurrent program by limiting resources.	One solution is to consider allocating additional resources. Another solution is to limit all threads' access to resources.
Incorrect count initialization anti-pattern.+	This pattern occurs when there is an incorrect initialization in a barrier for the number of parties that must be waiting for the barrier to trip, or an incorrect initialization of the number of threads required to complete some action in a latch, or an incorrect initialization of the number of permits in a semaphore.	Trying to protect access to operations involving shared data.	Correct the count to the appropriate value.



Pattern name	Problem	Context	Solution
Nonatomic operations assumed to be atomic anti-pattern.*	*"...an operation that "looks" like one operation in one programmer model (e.g., the source code level of the programming language) but actually consists of several unprotected operations at the lower abstraction levels"* [8].	Trying to perform an operation on a shared data variable atomically.	Use the volatile keyword when using 64-bit variables.
Two-state access bug anti-pattern.*	*"Sometimes a sequence of operations needs to be protected but the programmer wrongly assumes that separately protecting each operation is enough"* [8].	Trying to protect access to operations involving shared data.	Combine the multiple critical regions into one critical region.
Wrong lock or no lock bug anti-pattern.*	*"A code segment is protected by a lock but other threads do not obtain the same lock instance when executing. Either these other threads do not obtain a lock at all or they obtain some lock other than the one used by the code segment"* [8].	Trying to protect access to operations involving shared data.	Identify all accesses to shared data and use the same lock object to protect these critical regions. This may involve added a new lock or replacing incorrect locks with the correct one.
Double-checked lock anti-pattern.*	*"When an object is initialized, the thread local copy of the objects field is initialized but not all object fields are necessarily written to the heap. This might cause the object to be partially initialized while its reference is not null"* [8].	Trying to initialize shared variables without using protection.	Use locks to synchronize all access to the object or use volatile. Do not perform lazy initialization on shared objects.
The sleep() anti-pattern.*	*"The programmer assumes that a child thread should be faster than the parent thread in order that its results be available to the parent thread when it decides to advance. Therefore, the programmer sometimes adds an 'appropriate' sleep() to the parent thread. However, the parent thread may still be quicker in some environment."* [8].	Trying to coordinate threads based on assumptions regarding thread timing.	*"The correct solution would be for the parent thread to use the join() method to explicitly wait for the child thread"* [8].
Missing or nonexistent signals anti-pattern.+	This pattern generalizes the losing a notify bug pattern to all signals. The losing a notify bug is defined as occurring *"If a notify() is executed before its corresponding wait(), the notify() has no effect and is "lost" ... the programmer implicitly assumes that the wait() operation will occur before any of the corresponding notify() operations"* [8]. Another example of this problem can occur at a barrier. If an await() from one thread never occurs then all of threads at the barrier may be stuck waiting.	Trying to coordinate threads based on assumptions regarding thread timing.	*In the case of a notify signal*, "One way of avoiding this bug pattern is to repeatedly execute the notify() operation until a condition stating that the notify() was received occurs"[8]. Use concurrent mechanisms such as barriers and join() to prevent thread timing issues. Analogous solutions exist for other signals.
Notify instead of notify all anti-pattern.**	If a notify() is executed instead of notifyAll() then threads with some of its corresponding wait() calls will not be notified [16].	Trying to coordinate threads.	Replace notify() with notifyAll().
A "blocking" critical section anti-pattern.*	*"A thread is assumed to eventually return control but it never does"* [8].	Using locks to try and protect access to operations involving shared data.	Ensure that every lock() acquisition has a corresponding unlock(). If it is possible to throw an exception inside a critical region the unlock() must be placed in a finally block. The finally block will be executed regardless if the exception is thrown.



Pattern name	Problem	Context	Solution
Nonatomic operations assumed to be atomic anti-pattern.*	*"...an operation that "looks" like one operation in one programmer model (e.g., the source code level of the programming language) but actually consists of several unprotected operations at the lower abstraction levels"* [8].	Trying to perform an operation on a shared data variable atomically.	Use the volatile keyword when using 64-bit variables.
Two-state access bug anti-pattern.*	*"Sometimes a sequence of operations needs to be protected but the programmer wrongly assumes that separately protecting each operation is enough"* [8].	Trying to protect access to operations involving shared	Combine the multiple critical regions into one critical region.
Wrong lock or no lock bug anti-pattern.*	*"A code segment is protected by a lock but other threads do not obtain the same lock instance when executing. Either these other threads do not obtain a lock at all or they obtain some lock other than the one used by the code segment"* [8]	Trying to protect access to operations involving shared data.	Identify all accesses to shared data and use the same lock object to protect these critical regions. This may involve added a new lock or replacing incorrect locks with the correct one.
Double-checked lock anti-pattern.*	*"When an object is initialized, the thread local copy of the objects field is initialized but not all object fields are necessarily written to the heap. This might cause the object to be partially initialized while its reference is not null"* [8].	Trying to initialize shared variables without using protection.	Use locks to synchronize all access to the object or use volatile. Do not perform lazy initialization on shared objects.
The sleep() anti-pattern.*	*"The programmer assumes that a child thread should be faster than the parent thread in order that its results be available to the parent thread when it decides to advance. Therefore, the programmer sometimes adds an 'appropriate' sleep() to the parent thread. However, the parent thread may still be quicker in some environment."* [8].	Trying to coordinate threads based on assumptions regarding thread timing.	*"The correct solution would be for the parent thread to use the join() method to explicitly wait for the child thread"* [8].
Missing or nonexistent signals anti-pattern.+	This pattern generalizes the losing a notify bug pattern to all signals. The losing a notify bug is defined as occurring *"If a notify() is executed before its corresponding wait(), the notify() has no effect and is "lost" ... the programmer implicitly assumes that the wait() operation will occur before any of the corresponding notify() operations"* [8]. Another example of this problem can occur at a barrier. If an await() from one thread never occurs then all of threads at the barrier may be stuck waiting.	Trying to coordinate threads based on assumptions regarding thread timing.	*In the case of a notify signal,* "One way of avoiding this bug pattern is to repeatedly execute the notify() operation until a condition stating that the notify() was received occurs"[8]. Use concurrent mechanisms such as barriers and join() to prevent thread timing issues. Analogous solutions exist for other signals.
Notify instead of notify all anti-pattern.**	If a notify() is executed instead of notifyAll() then threads with some of its corresponding wait() calls will not be notified [16].	Trying to coordinate threads.	Replace notify() with notifyAll().
A "blocking" critical section anti-pattern.*	*"A thread is assumed to eventually return control but it never does"* [8].	Using locks to try and protect access to operations involving shared data.	Ensure that every lock() acquisition has a corresponding unlock(). If it is possible to throw an exception inside a critical region the unlock() must be placed in a finally block. The finally block will be executed regardless if the exception is thrown.

Example: Wrong lock or no lock anti-pattern

- Problem:
  A code segment is protected by a lock but other threads do not obtain the same lock instance when executing. Either these other threads do not obtain a lock at all or they obtain some lock other than the one used by the code segment. [FNU03]
- Context: Trying to protect access to operations involving shared data.
- Solution: Identify all accesses to shared data and use the same lock object to protect these critical regions. This may involve added a new lock or replacing incorrect locks with the correct one.

[FNU03] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in Proc. of the 1st Int. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003), Apr. 2003.

Example: Wrong lock or no lock anti-pattern

```
Object lock1  = new Object();
...
public void m1 () {
   <statement n1>
   //critical region
   <statement c1>
   synchronized (lock1) {
       <statement c2>
   }
   <statement c3>
   <statement n2>
...
```

```
Object lock1  = new Object();
...
public void m1 () {
   <statement n1>
  synchronized (lock1) {
       //critical region
       <statement c1>
       <statement c2>
       <statement c3>
  }
  <statement n2>
...
```

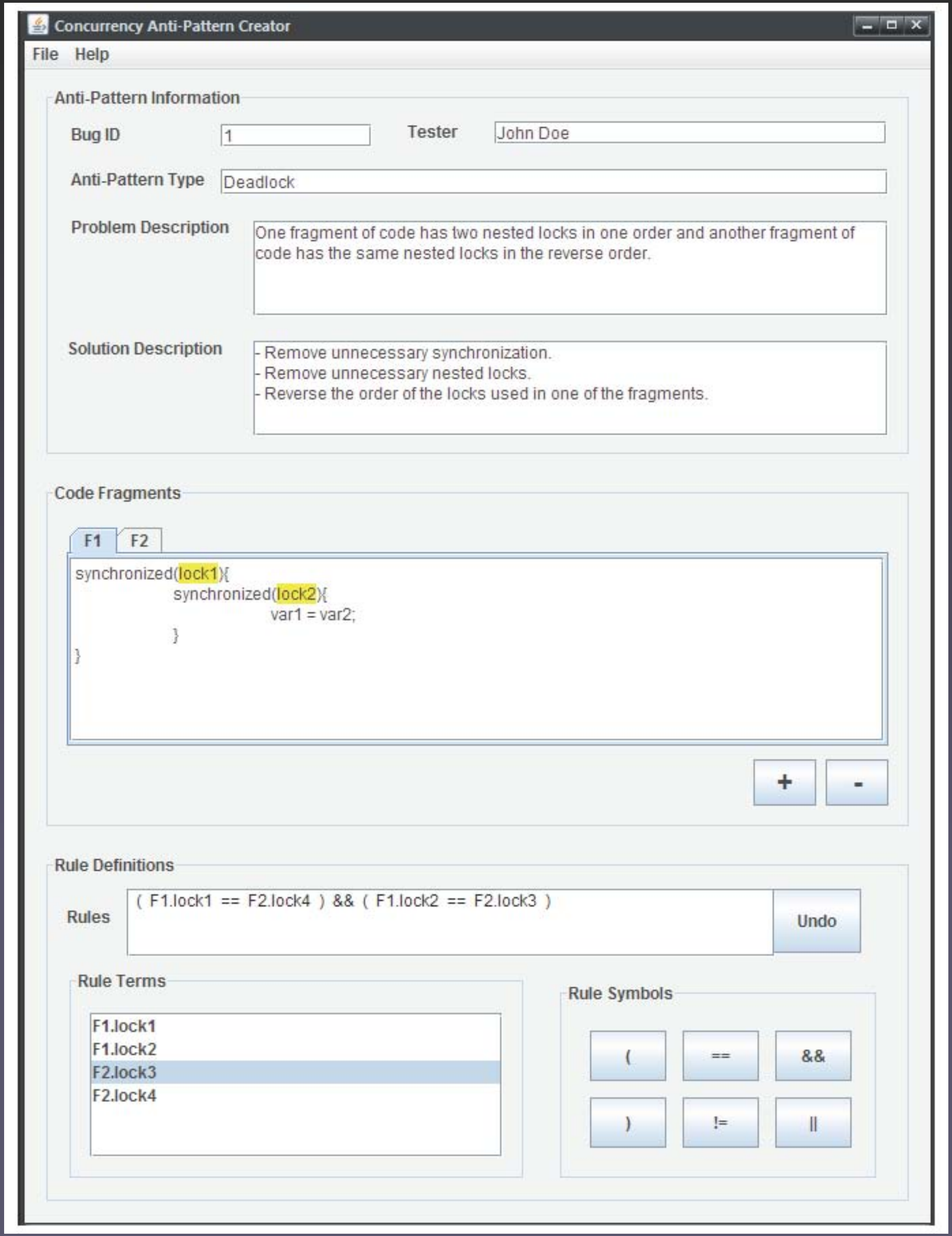
Detecting Concurrency Anti-Patterns

- The main motivation for our work on concurrency anti-patterns has been the automatic detection (and correction)

- We have developed a combined approach to static analysis and testing that uses our concurrency anti-patterns to *focus* the testing effort

Concurrency Anti-Pattern Creator

- A tool for the creation and storage of concurrency anti-patterns



Bug ID: 1
Tester: John Doe
Anti-Pattern Type: Deadlock
Problem Description: One fragment of code has two nested locks in one order and another fragment of code has the same nested locks in the reverse order.
Solution Description:
- Remove unnecessary synchronization.
- Remove unnecessary nested locks.
- Reverse the order of the locks used in one of the fragments.

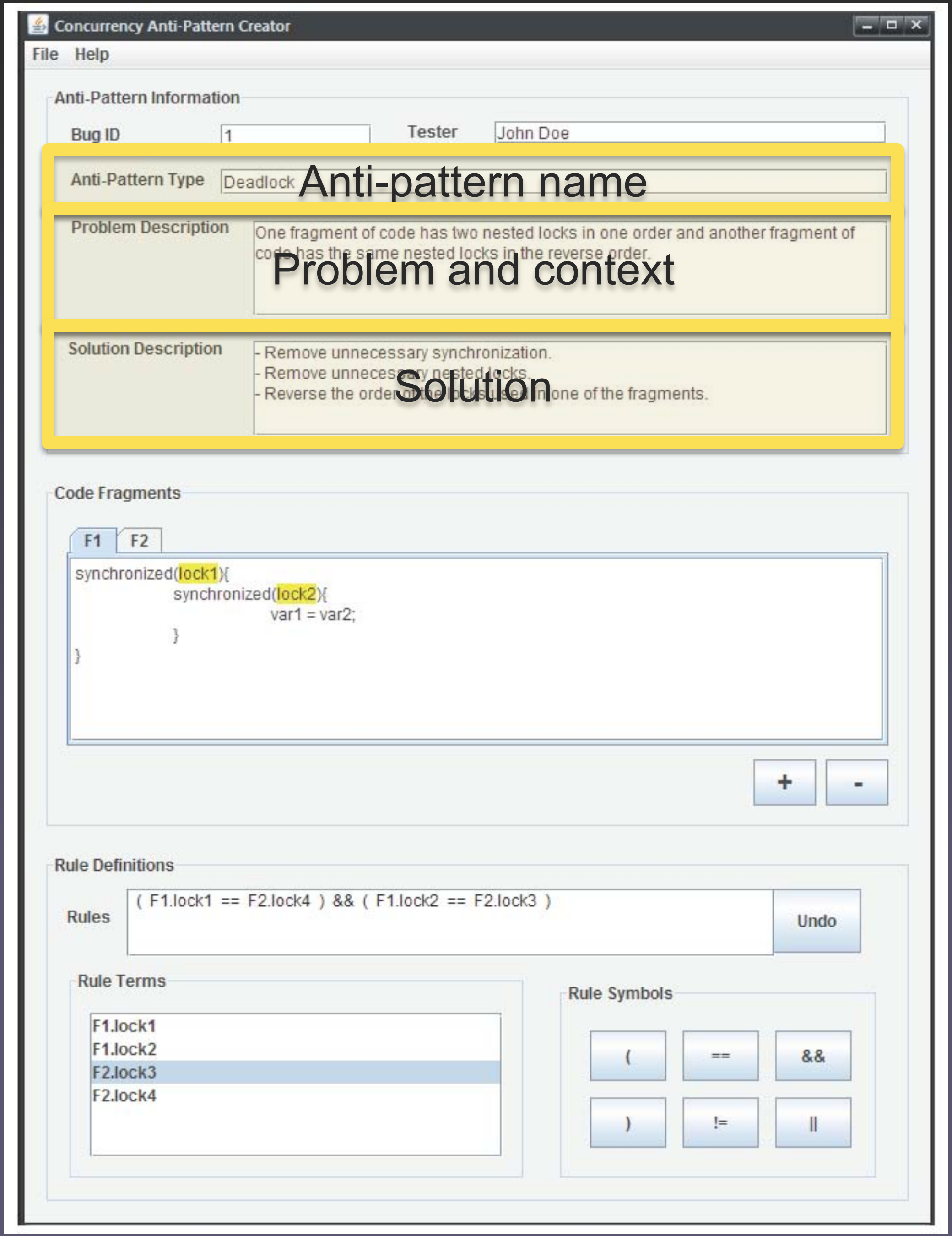
```
synchronized(lock1){
        synchronized(lock2){
                var1 = var2;
        }
}
```

Rules: (F1.lock1 == F2.lock4) && (F1.lock2 == F2.lock3)

F1.lock1
F1.lock2
F2.lock3
F2.lock4

Concurrency Anti-Pattern Creator

- A tool for the creation and storage of concurrency anti-patterns



Anti-Pattern Information

Bug ID: 1 Tester: John Doe

Anti-Pattern Type: Deadlock

Anti-pattern name

Problem Description: One fragment of code has two nested locks in one order and another fragment of code has the same nested locks in the reverse order.

Problem and context

Solution Description:
- Remove unnecessary synchronization.
- Remove unnecessary nested locks.
- Reverse the order of the locks used in one of the fragments.

Solution

Code Fragments

```
synchronized(lock1){
        synchronized(lock2){
                var1 = var2;
        }
}
```

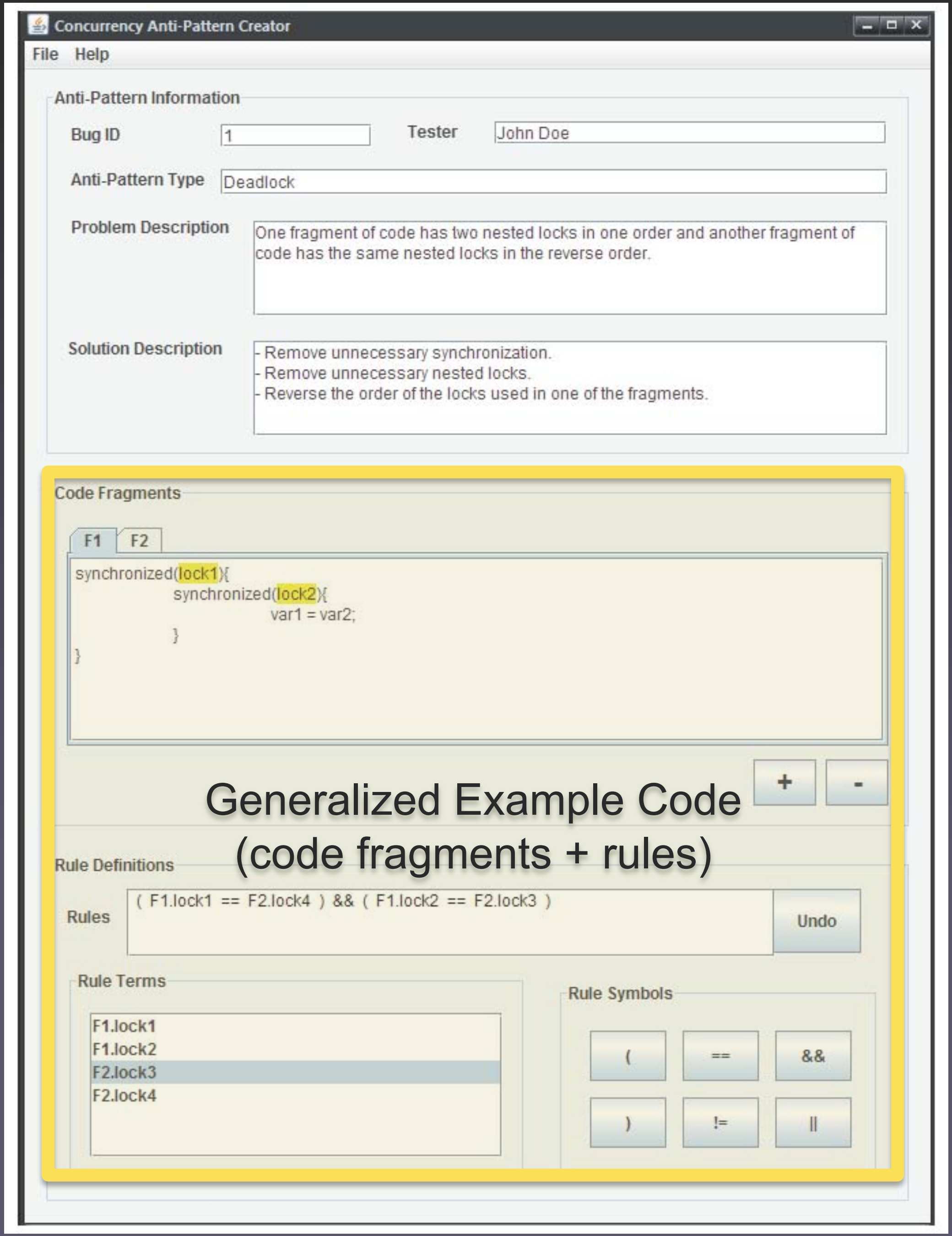
Rule Definitions

Rules: (F1.lock1 == F2.lock4) && (F1.lock2 == F2.lock3)

Rule Terms
- F1.lock1
- F1.lock2
- F2.lock3
- F2.lock4

Concurrency Anti-Pattern Creator

- A tool for the creation and storage of concurrency anti-patterns



Concurrency Anti-Pattern Creator

File Help

Anti-Pattern Information

Bug ID	1	Tester	John Doe
Anti-Pattern Type	Deadlock		
Problem Description	One fragment of code has two nested locks in one order and another fragment of code has the same nested locks in the reverse order.		
Solution Description	- Remove unnecessary synchronization. - Remove unnecessary nested locks. - Reverse the order of the locks used in one of the fragments.		

Code Fragments

F1 F2

```
synchronized(lock1){
        synchronized(lock2){
                var1 = var2;
        }
}
```

Generalized Example Code (code fragments + rules)

Rule Definitions

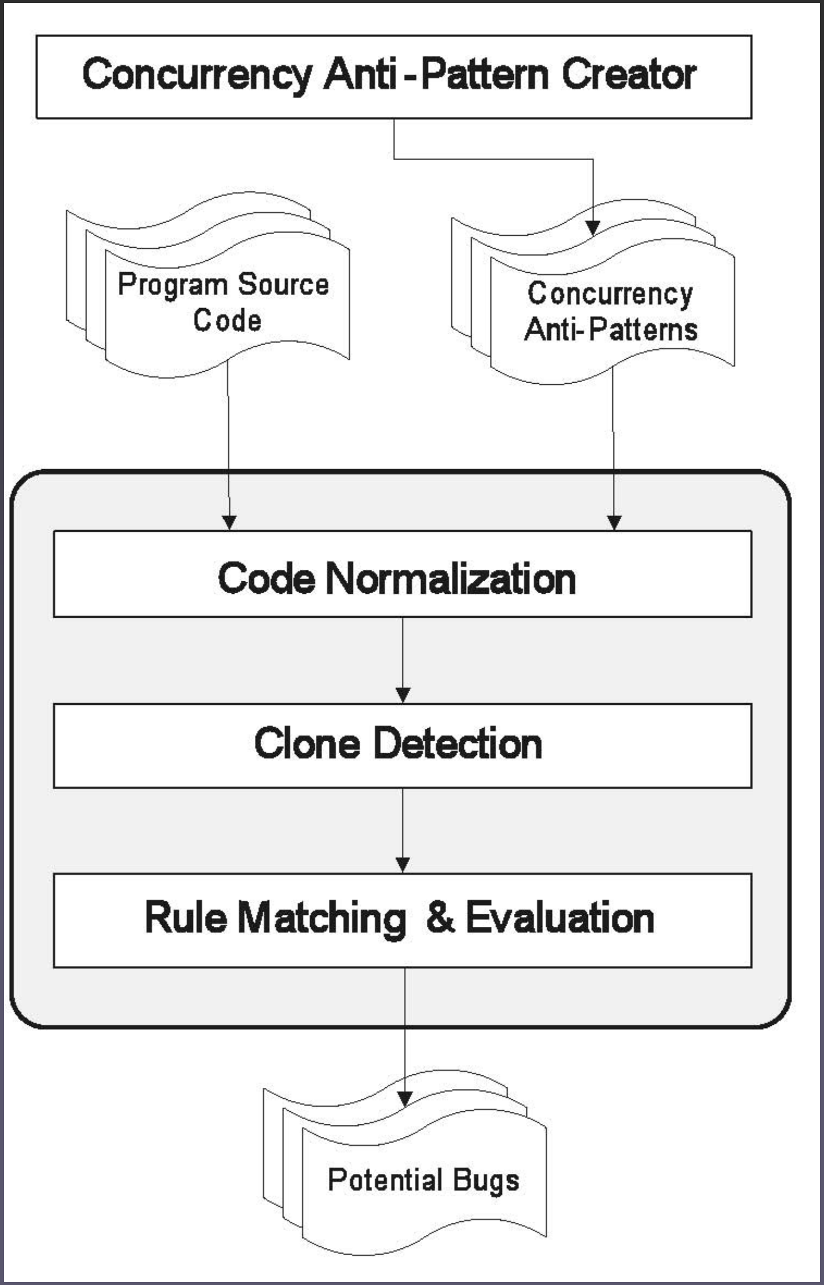
Rules: (F1.lock1 == F2.lock4) && (F1.lock2 == F2.lock3)

Rule Terms

- F1.lock1
- F1.lock2
- F2.lock3
- F2.lock4

Clone-based Detection Tool

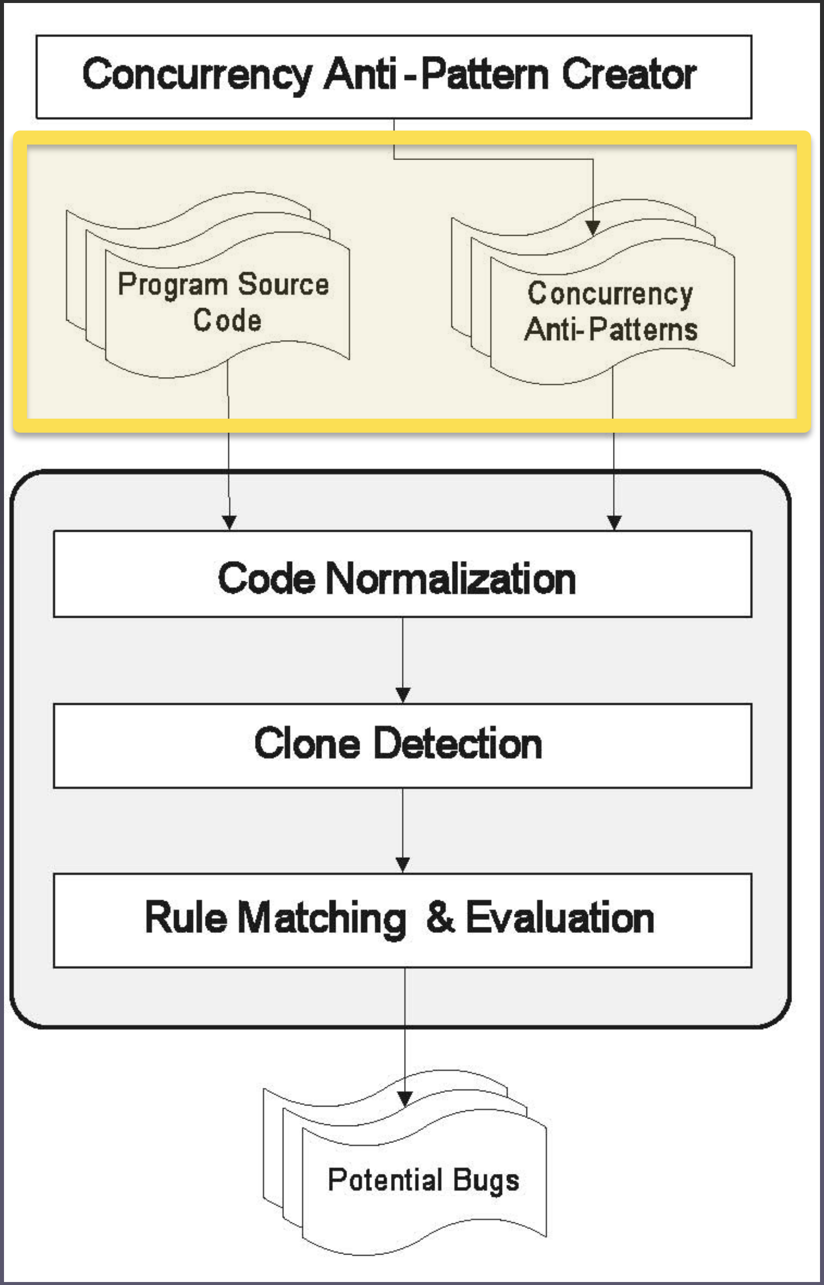
- A tool for the detecting concurrency anti-patterns in Java programs
- Built using the pattern matching algorithm in the ConQAT* clone detection tool


Concurrency Anti-Pattern Creator
Program Source Code
Concurrency Anti-Patterns
Code Normalization
Clone Detection
Rule Matching & Evaluation
Potential Bugs

* ConQAT website (http://conqat.cs.tum.edu/)

Clone-based Detection Tool

- A tool for the detecting concurrency anti-patterns in Java programs
- Built using the pattern matching algorithm in the ConQAT* clone detection tool

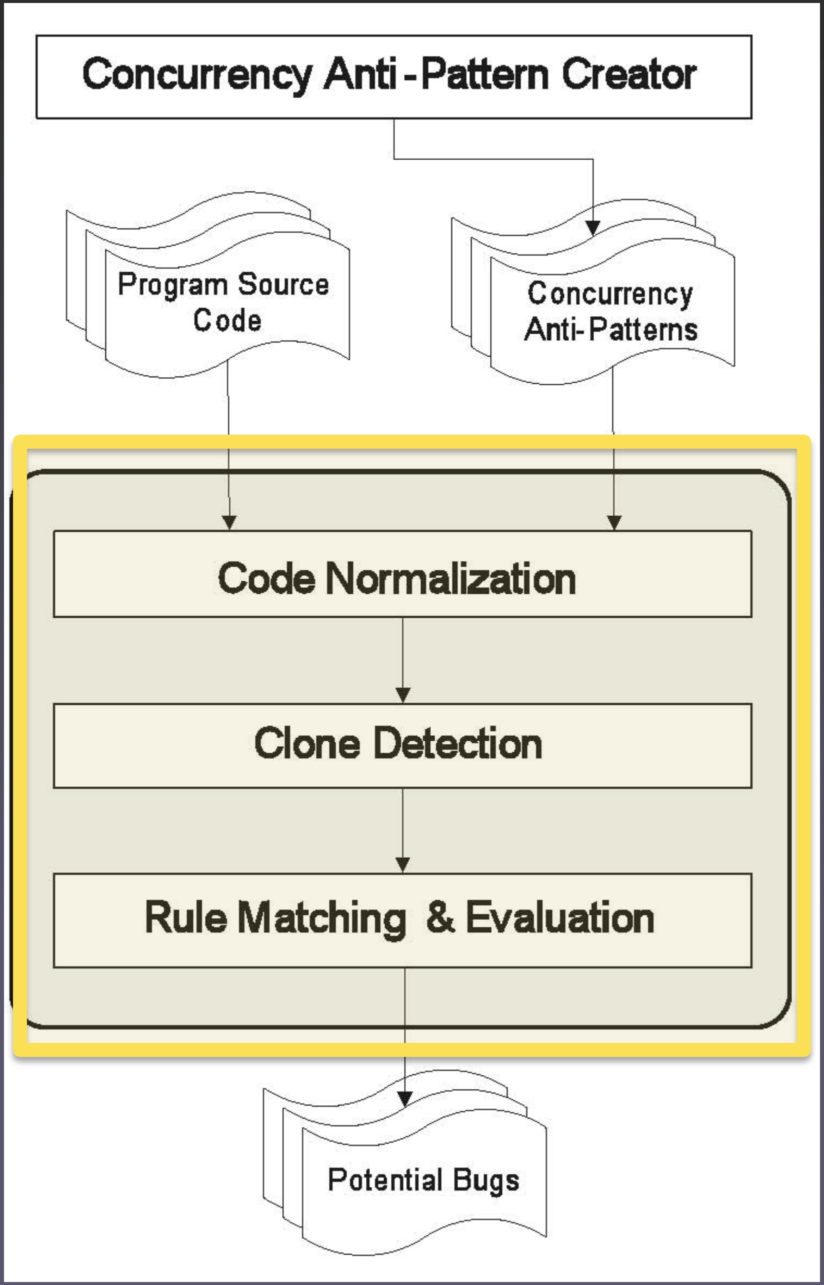

Concurrency Anti-Pattern Creator
Program Source Code
Concurrency Anti-Patterns
Code Normalization
Clone Detection
Rule Matching & Evaluation
Potential Bugs

* ConQAT website (http://conqat.cs.tum.edu/)

Clone-based Detection Tool

- A tool for the detecting concurrency anti-patterns in Java programs
- Built using the pattern matching algorithm in the ConQAT* clone detection tool

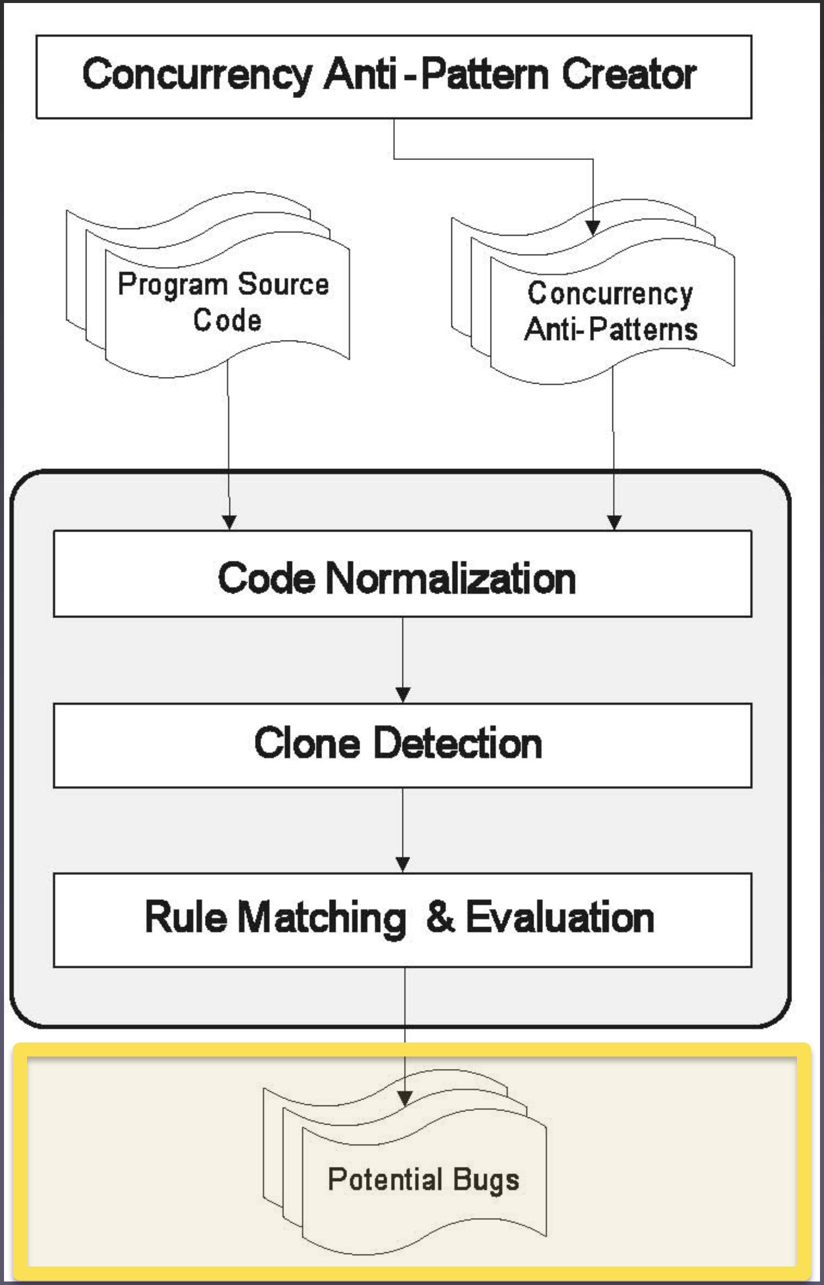
* ConQAT website (http://conqat.cs.tum.edu/)



Concurrency Anti-Pattern Creator
Program Source Code
Concurrency Anti-Patterns
Code Normalization
Clone Detection
Rule Matching & Evaluation
Potential Bugs

Clone-based Detection Tool

- A tool for the detecting concurrency anti-patterns in Java programs
- Built using the pattern matching algorithm in the ConQAT* clone detection tool


Concurrency Anti-Pattern Creator
Program Source Code
Concurrency Anti-Patterns
Code Normalization
Clone Detection
Rule Matching & Evaluation
Potential Bugs

* ConQAT website (http://conqat.cs.tum.edu/)

Summary

- We have presented a catalog of 13 programming anti-patterns for concurrent Java that are comprehensive with
- respect to:
  - the Java concurrency features
  - an existing concurrency bug pattern taxonomy
- Catalog available at:

  http://svilab.science.uoit.ca/concurr-catalog/

- One important contribution of this work is that the catalog provides solutions – previous work has focused on enumerating different kinds of concurrency

Future Work

- We plan to conduct additional research on the benefits of the catalog with respect to static analysis and testing.
- We are interested in combining our work with more high-level concurrency design patterns [GHJV95], [Lea00] and other concurency anti-patterns [HAT+04].

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, ser. Addison-Wesley Professional Computing Series. Addison Wesley, 1995.

[Lea00] D. Lea, Concurrent Programming in Java: Design Principles and Patterns, Second Edition. Addison Wesley, 2000.

[HAT+04] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, and A. Petrenko, "Antipattern-based detection of deficiencies in Java multithreaded software," in 4th International Conference on Quality Software (QSIC 2004), 2004, pp. 258–267.

Defining a Catalog of Programming Anti-Patterns for Concurrent Java

Jeremy S. Bradbury, Kevin Jalbert
Software Quality Research Group
Faculty of Science (Computer Science)
University of Ontario Institute of Technology
Oshawa • Ontario • Canada
jeremy.bradbury@uoit.ca, kevin.jalbert@mycampus.uoit.ca

SPAQu'09 • October 25, 2009

SQR GROUP

SUPPORTED BY NSERC CRSNG