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

***Abstract*—Many programming languages, including Java, provide support for concurrency. Although concurrency has many benefits with respect to performance, concurrent software can be problematic to develop and test because of the many different thread interleavings. We propose a comprehensive set of concurrency programming anti-patterns that can be used by Java developers to aid in avoiding many of the known pitfalls associated with concurrent software development. Our concurrency anti-patterns build upon our previous work as well as the work of others in the research community.**

***Keywords*-concurrency, anti-patterns, bug patterns, Java, deadlock, race conditions, static analysis.**

I. Introduction

The widespread adoption of multi-core technologies has made concurrency an essential characteristic of many traditionally sequential programs. The use of concurrency with multi-core systems can provide an increase in performance over sequential code because it allows programs to have multiple threads executing simultaneously. Although concurrency is beneficial, it can also be problematic. For example, the possibly many different ways to interleave threads in concurrent code make it very difficult to test. Concurrency bugs can be hard to find due to the non-deterministic nature of thread interleavings and because some bugs may occur in only a small subset of the entire interleaving space. It is also challenging to reproduce these bugs and determine if a bug has been fixed or not. In general, concurrency bugs exhibit consequences not present in sequential source code, including deadlock and race conditions. These consequences typically occur because of problems with accessing shared data or controlling access to shared data.

In an effort to improve the quality of concurrent programs there has been considerable effort invested by researchers in developing new programming models, new testing and analysis tools and in identifying concurrency-related design patterns. The development of new concurrent programming models [1] has the potential to make programming with concurrency easier and less error prone. The development of new testing and analysis techniques, as well as the improvement of existing techniques, is aimed at identifying more concurrency bugs prior to deployment. The identification of concurrency design patterns complements the previous two research topics by focusing on how to improve concurrency programming in *existing* languages in an effort to *reduce* bugs prior to testing and analysis.

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"* [2]. A pattern should include details such as the pattern name, the problem, the solution to the problem and the consequences of using the pattern [3]. Alternatively, an anti-pattern defines a recuring bad design solution [4]. The goal of our research is to produce a set of low-level anti-patterns for improving concurrent source code. We have focused our efforts on low-level anti-patterns to complement the previous work on identify concurrency design-level patterns [5], [6].

In Section II we will discuss how to write concurrent programs in Java. We present our concurrency anti-patterns catalog in Section III and provide an example to illustrate the use of the catalog. We also discuss how our catalog can be used to improving concurrency programming, testing and analysis. In Section IV we address how to automatically detect potential anti-patterns in source code before presenting our conclusions in Section V.

II. Java Concurrency

Concurrent Java programs are often called multi-threaded programs. During execution an active thread can be runnable or not runnable and a number of methods exist that can affect a thread's status:

- sleep(): will cause the current thread to stop executing for a certain amount of time.
- yield(): will cause the current thread that is running to pause and yield the processor to another thread.
- 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 notify() or notifyAll().

Prior to J2SE 5.0, Java provided support for concurrency 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. In J2SE 5.0, additional mechanisms to support concurrency were added as part of the `java.util.concurrent` package [7]:

- Explicit Lock: Provides the same semantics as the implicit monitor locks but provides 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.

To reduce the overhead of developing concurrent software J2SE 5.0 also provides a number of other resources:

- Concurrent collection types: ConcurrentHashMap, BlockingQueues.
- Built-in thread pools: FixedThreadPool and an unbounded CachedThreadPool.
- Atomic variable types: Types that can be used in place of synchronization since each atomic variable type contains special atomic methods. For example, AtomicInteger contains a methods getAndSet().

III. A Catalog of Concurrency Anti-Patterns

Prior to J2SE 5.0, Farchi, Nir, and Ur developed a bug pattern taxonomy for Java concurrency [8]. The bug patterns are based on common mistakes programmers make when developing concurrent code in practice. Furthermore, the taxonomy has been expanded and used to classify bugs in an existing public domain concurrency benchmark maintained by IBM Research [9]. Bradbury, Cordy and Dingel further extended the taxonomy in their concurrency mutation research [10]. We will use this bug taxonomy as the basis for our concurrency anti-patterns – in fact many of the problems we identify were included in this previous work.

An anti-pattern catalog for Java multithreaded software has already been developed by Hallal et al. [6]. In their work, Hallal et al. distinguish between design anti-patterns and error or bug patterns. The former category focuses on the syntactic design within a program while the latter category focuses on *"patterns of erroneous program behavior correlated with programming mistakes"* [6]. The Hallal et al. anti-pattern catalog primarily contains design anti-patterns, including anti-patterns related to efficiency, quality and style, while 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 the Hallal et al. catalog and our catalog are complementary.

Table I and II provide an overview of all the concurrency anti-patterns included in our catalog[1]. For each anti-pattern we provide the following information:

- *pattern name:* the anti-pattern name is based on the corresponding bug's name. For example, the two-state access anti-pattern corresponds to the two-state access bug.
- *problem:* the problem describes the corresponding bug that is being addressed.
- *context:* the context in which the problem often occurs.
- *solution:* the solution describes general steps that can be taken to correct the anti-pattern. We have made an effort to keep the solutions as general as possible and it is expected that the developer will have the appropriate level of knowledge to understand how to apply the solution in a specific context.

We have not included the consequences of fixing each anti-pattern because in most cases these are evident from the problem section of the anti-pattern. For example, the consequences of applying the solution in the *Deadlock anti-pattern* are that locks will now be released and the threads will no longer halt.

Our catalog of concurrency anti-patterns provides several benefits:

1) The catalog is language specific – it is focused on anti-patterns that can occur in Java and not anti-patterns that occur in general.
2) The catalog is comprehensive – it includes the bug definitions from several different sources [8], [9], [10].
3) The catalog provides solutions – in addition to enumerating different kinds of concurrency bugs as anti-pattern problems, we also provide solutions to each anti-pattern.

To demonstrate the use of the catalog we will now describe an example using the *Deadlock anti-pattern*. Consider the following two code fragments which are executed by different threads:

Code fragment #1:

```
public void methodA(){
   synchronized(lock1){
      synchronized(lock2){    }
   }
}
```

Code fragment #2:

```
public void methodB(){
   synchronized(lock3){
      synchronized(lock4){    }
   }
}
```

[1]In Table I and II we distinguish between the original bugs from [8] (*), the added bug used in the benchmark classification [9] (**) and the bugs included in [10] ($^{+}$).

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.

Table I
CONCURRENCY ANTI-PATTERNS CATALOG *(Part 1 of 2)*

The above fragments are an example of the *Deadlock anti-pattern* if lock1 is the same lock object as lock4 while lock2 is the same lock object as lock3. If the above fragments are an example of the *Deadlock anti-pattern* then we have several

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.

Table II
CONCURRENCY ANTI-PATTERNS CATALOG *(Part 2 of 2)*

options regarding how to correct the code fragments. First we need to ensure that both locks are indeed necessary. If any lock is not necessary it should be removed. If all locks are necessary, we next consider whether the locks need to be nested. If not we rewrite the code to separate the critical region into two separate regions each protected by one of the locks. If nested synchronization is necessary, we need to ensure that the lock objects are always acquired in the same order. This example illustrates how a catalog of concurrency anti-patterns can aide in improving the quality of concurrent software. We believe that the benefits of this work fall into three key areas: programming, testing and static analysis.

Programming. There are many examples of software design patterns that have been adopted and used in industry [3], [5]. A benefit of these patterns is that they clearly show good ways (or in the case of anti-patterns bad ways) to design or implement software. The goal of our concurrency anti-patterns is to provide programmers an additional resource that will assist them in concurrent Java programming by sharing potential problems that should be avoided. The benefit of our anti-patterns is that they help to clarify bad concurrency practices which can assist developers in avoiding concurrency bugs and thus result in improved source code.

Testing. Sequential testing typically involves developing a set of test cases that provide a certain type of code coverage (e.g., path coverage) and executing these tests on the code to detect possible bugs and failures. Due to the non-determinism of the execution of concurrent source code and the high number of possible interleavings, concurrency testing can not rely on coverage metrics alone to guarantee that code is correct. Concurrency testing must also provide increased confidence that bugs that manifest themselves in only a few of the interleavings are found. For example,

since a race condition or deadlock may only occur in a small subset of the possible interleaving space, the more interleavings we test the higher our confidence that the bug that caused the race condition or deadlock will be found. An example of a tool for executing different thread schedules is ConTest [11].

A catalog of concurrency anti-patterns can benefit concurrency testing by helping to direct the testing effort. A good understanding of concurrency bugs can provide a tester with more insight into the problems he or she may encounter as well as help a tester focus his or her testing effort within the interleaving space.

Static analysis. Static analysis can be used throughout the software development life cycle and provides useful information about the possible presence of bugs in software. For example, a static analysis tool that detects a match of the *Deadlock anti-pattern* may help a programmer improve his or her code during implementation or may help catch a bug during testing. Existing static analysis tools, including FindBugs [12], JLint [13] and the Otto-Moschny tool [14], already utilize some concurrency bug patterns in an effort to identify potential problems in concurrent Java source code. Our catalog of concurrency anti-patterns will aide in improving existing tools as well as in the development of new static analysis tools.

IV. Detecting Concurrency Anti-Patterns

In addition to developing our concurrency anti-pattern catalog we have also developed several tools to assist programmers in managing anti-patterns and in identifying potential anti-pattern matches in source code.

A. Concurrency Anti-Pattern Creator

Concurrency anti-patterns can be created and managed using the Concurrency Anti-Pattern Creator tool (see Figure 1). In this tool we define a concurrency anti-pattern as consisting of a name, a problem (with context), a solution, one or more fragments of code as well as a rule about how the fragments interact to cause undesired behaviour. Our experience has shown that many concurrency bugs result in a combination of different code fragments executing in different threads. The interaction of code fragments from different threads is specified in the anti-pattern definition using a rule. Specifically, the rule explains how the code fragments interact to produce erroneous output.

B. Clone-Based Detection of Concurrency Anti-Patterns

One of our goals in creating our concurrency anti-pattern catalog was to also create a static analysis tool to detect possible anti-pattern matches. In our tool, a potential match in source code is made to a known anti-pattern only if all code fragments are present and the rule is satisfied. Our approach takes program source code and anti-patterns as input. The source code is normalized and input to the clone

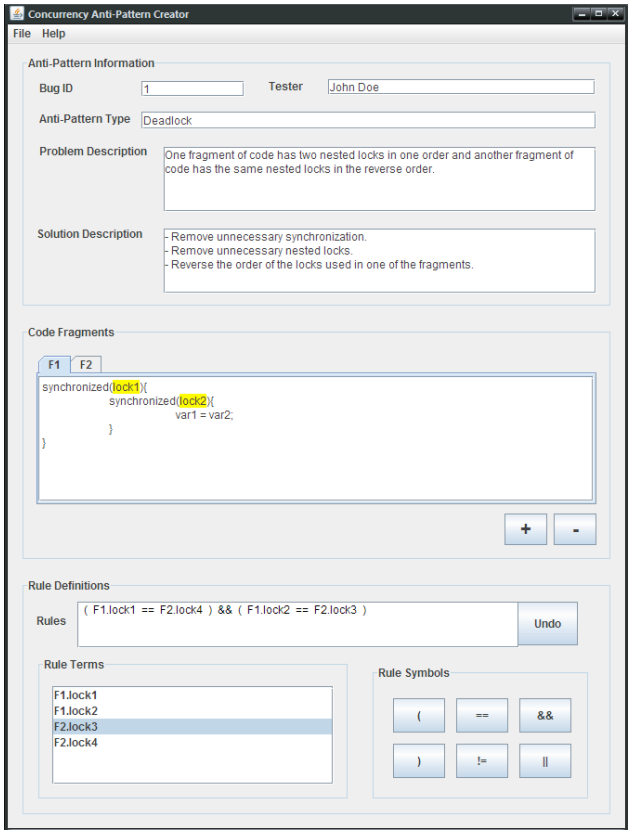

Figure 1. Concurrency Anti-Pattern Creator

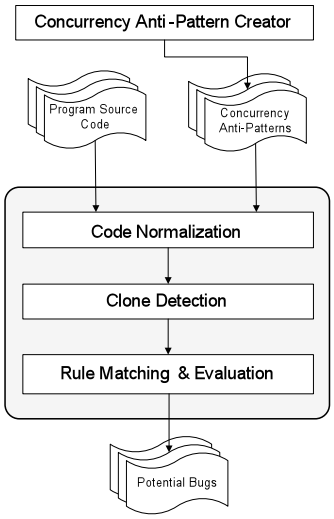

Figure 2. Detection of anti-pattern matches in source code

detection tool ConQAT [15]. We use the pattern matching in ConQAT to determine if code fragments in the source code match the code fragments in our anti-pattern. In cases where ConQAT finds matches for all code fragments in an

anti-pattern we use rule matching to determine if a particular combination of code fragments satisfy the anti-pattern rule. If the rule is satisfied we identify the code fragments as a potential match to our anti-pattern. At this point the developer can use the anti-pattern catalog to determine the appropriate fix (if the match is not a false positive).

An important feature of our detection tool is that it is designed to detect any anti-pattern specified using the Concurrency Anti-Pattern Creator. This feature ensures that the detection tool is flexible enough to be extended to any future anti-patterns that could be added to the catalog. It also means that the catalog can be customized to a particular project or source code repository.

V. Conclusion

In this paper we have presented a catalog of programming anti-patterns for concurrent Java that are comprehensive with respect to the programming features available in the Java programming language and comprehensive with respect to an existing concurrency bug pattern taxonomy. We will be making our catalog available publicly[2] and providing the community an opportunity to both use and contribute anti-patterns.

In the future we hope to conduct additional research on the benefits of the catalog with respect to static analysis and testing. We are also interested in studying how these anti-patterns can be utilized in combination with more high-level design patterns [3], [5] and the Hallal et al. anti-patterns [6].

Acknowledgment

The authors would like to thank Shmuel Ur for providing access to IBM Haifa Lab's Concurrency Bug Benchmark. We would also like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for funding this research.

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*. Oxford University Press, 1977.

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

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

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

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

[7] "`java.util.concurrent` documentation," Web page: http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html.

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

[9] Y. Eytani and S. Ur, "Compiling a benchmark of documented multi-threaded bugs," in *Proc. of the* 2^{nd} *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, Apr. 2004.

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

[11] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation." *IBM Systems Journal*, vol. 41, no. 1, pp. 111–125, 2002.

[12] "Findbugs - find bugs in Java programs," Web page: http://findbugs.sourceforge.net/.

[13] *Jlint Manual: Java program checker*, Web page: http://artho.com/jlint/manual.html, Jan. 2002.

[14] F. Otto and T. Moschny, "Finding synchronization defects in java programs: extended static analyses and code patterns," in *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*. New York, NY, USA: ACM, 2008, pp. 41–46.

[15] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka, "Tool support for continuous quality control," *IEEE Software*, vol. 25, no. 5, pp. 60–67, 2008.

[16] B. Long, R. Duke, D. Goldson, P. A. Strooper, and L. Wildman, "Mutation-based exploration of a method for verifying concurrent Java components," in *Proc. of the* 2^{nd} *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, Apr. 2004.

[17] B. Long, P. Strooper, and L. Wildman, "A method for verifying concurrent Java components based on an analysis of concurrency failures," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 281–294, Mar. 2007.

[18] C. Artho, K. Havelund, and A. Biere, "High-level data races," in *Proc. of the* 1^{st} *International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03)*, Apr. 2003.

[19] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, 1992.

[2] http://svilab.science.uoit.ca/concurr-catalog/