Concurrent Bug Patterns and How to Test Them

Eitan Farchi, Yarden Nir, Shmuel Ur
Verification Technologies Department
IBM Research Lab in Haifa
Haifa University Campus,
Mount Carmel, Haifa 31905, Israel
{farchi, nir, ur}@il.ibm.com

Abstract

We present and categorize a taxonomy of concurrent bug patterns. We then use the taxonomy to create new timing heuristics for ConTest. Initial industrial experience indicates that these heuristics improve the bug finding ability of ConTest. We also show how concurrent bug patterns can be derived from concurrent design patterns. Further research is required to complete the concurrent bug taxonomy and formal experiments are needed to show that heurisitics derived from the taxonomy improve the bug finding ability of ConTest.

1 Introduction

A bug taxonomy for sequential programs was used in [1] to motivate test techniques. A short bug taxonomy for concurrent programs was described in [9] for the POSIX pthread library, but no test techniques were developed based on this taxonomy. This paper describes and categorizes a more detailed taxonomy of concurrent bugs. We then use the taxonomy to create new heuristics for ConTest [4]. For precision, the bugs are described in the context of the Java programming language, but most of them are not programming language specific.

Design patterns [5] are solutions to recurring problems in a given context. A design pattern accentuates the positive, i.e., how to solve a recurring problem well. In the software design process, recurring problems are repeatedly solved in the wrong way. To describe this, the concept of an "Antipattern" was introduced. An antipattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences [2]. As such, an antipattern accentuates the negative.

Errors are introduced throughout the software development cycle. At the implementation level, these errors are referred to as bugs. A bug pattern is an abstraction of a recurring bug. In other words, a bug pattern is a literary form that describes a commonly occurring error in the implementation of the software design.

Eric Allen regularly describes bug patterns at the developerWorks site [1]. His work contains some concurrent bug patterns, i.e., bug patterns in concurrent programs. In this paper, we use the concept of concurrent bug patterns to develop our taxonomy. There can be two different types of bug patterns: those that involve an error introduced when implementing a design pattern that solves the problem well, and those that involve an error introduced when implementing an antipattern.

The ConTest tool [4] and Scott D. Stoller's more recent work [11] represent a new technology aimed at increasing the chance of revealing concurrent bug patterns by changing the timing of the run of a concurrent program. A concurrent event [3] is a shared memory access or some synchronization event. A timing heuristic is an algorithm that sometimes forces context switches at concurrent events based on some decision function. We describe how ConTest was enhanced by introducing new heuristics that, based on our industrial experience, increase the chance of finding some of the concurrent bug patterns described in this paper. Formal experiments are needed to show that heuristics derived from the taxonomy actually improve the bug finding ability of ConTest.

This paper is organized as follows. In the second section we introduce and categorize different concurrent bug patterns that occur in practice. In the third section we describe several new timing heuristics motivated by concurrent bug patterns. In the fourth section we demonstrate how design patterns can be used to derive concurrent bug patterns. We then conclude and discuss future work.

[1] See $http\ //www.j2life.com/bitterjava/resources.html.$

2 Concurrent Bug Patterns

As mentioned in the introduction, the concept of a concurrent bug pattern is not new. Most of the general categories and most of the bug patterns introduced in this section are new. When appropriate, references are given to previous work. To facilitate the understanding and categorization of concurrent bug patterns, some definitions are required.

2.1 Overview and Definitions

For a run of a given concurrent program, its interleaving is the sequence of concurrent events that occurred in this run [2]. Thus, a concurrent program P is associated with a set of possible interleavings $I(P)$, i.e., the set of interleavings that can occur in some run of the concurrent program P. One way of viewing a concurrent bug pattern of a given concurrent program P is to look at the relation of the space of possible interleavings, $I(P)$, to the maximal space of interleavings for P under which the program is correct, $C(P)$. A concurrent bug pattern can be viewed as defining interleavings in $I(P) - C(P)$. We will sometimes refer to $C(P)$ as the $programmerview$ of the program P and to $I(P) - C(P)$ as the $bugpatterngap$.

For example, consider a nonatomic, two byte, integer write operation. For clarity, we deviate from precise Java syntax and semantics. At the source code level, this operation might look like:

$int\ x;\ x = 257;$

In contrast, at the object level, the operation could be broken down into two separate one-byte write operations, e.g.,

$x[0] = 1; // set\ the\ lower\ byte\ to\ 1$
$x[1] = 1; // set\ the\ upper\ byte\ to\ 1,$
$\quad // thus\ the\ value\ of\ x\ is\ 256 + 1\ =\ 257$

Consider two threads executing concurrently as follows:

$x = 257\ ||\ x = 0$

Then the space of possible correct interleavings, $C(P)$, or the "programmer view," includes the two assignments, in any order, with a final value of 257 or 0. Actually, a context switch might occur after the first $x[0] = 1$ is executed. This would result in the following interleaving:

$x[0] = 1;\ //first\ thread,\ assignment\ to\ lower\ byte$
$x = 0; //\ second\ thread$
$x[1] = 1;\ //first\ thread\ again,\ assignment\ to$
$\quad //\ higher\ byte$

with the result of $x = 256$ at the end of the run. The last statement and the use of the term "context switch" is justified regardless of whether the underlying machine is a single or multiple CPU machine, as long as replay [3] is possible.

A context switch that occurs between the assignments to $x[0]$ and $x[1]$ defines an interleaving in the set $I(P) - C(P)$ of incorrect interleavings. In other words, the implicit atomicity assumption of a nonatomic operation defined the $I(P) - C(P)$ bug pattern gap in this example.

Generally speaking, the following high level categories of concurrent bug patterns can be identified. We believe these categories to be reasonably complete. In contrast, the examples in the next section are only preliminary and possibly another level of categorization is needed.

- An interleaving in $I(P) - C(P)$ is created when a code segment is mistakenly assumed to be undisturbed, implicitly or explicitly, by other threads. (See the definition of undisturbed threads in the next section.) The example of a nonatomic operation assumed to be atomic is one such concurrent bug.
- An interleaving in $I(P) - C(P)$ is created as a result of the mistaken assumption that a certain execution order of concurrent events is impossible. This category is also mentioned in [9].
- An interleaving in $I(P) - C(P)$ is created when a code segment is mistakenly assumed to be nonblocking. In this case an interleaving executing the blocking code is in $I(P) - C(P)$.

A deadlock occurs when there is a cycle in the graph of resource acquisitions [6]. Since this concurrent bug pattern has been thoroughly discussed in the literature [6], we do not discuss it here.

The above three categories overlap. Note that these three categories overlap. This is fine, as they are intended to broadly describe different ways in which programmers make mistakes. An example of how the first and the third categories overlap follows. As a result of the wrong assumption that a code segment is assumed to be undisturbed, the programmer might reach the wrong conclusion that this code segment can never block. To be concrete, assume that a code segment can block if a server thread is not in a ready state. The programmer checks that the server thread is in the ready state before attempting to execute the code segment. Now if the programmer makes a mistake by assuming that the code segment will not be disturbed after the check,

[2]Given the serialization assumption, we can reasonably assume that there is an abstraction level in which a concurrent program can be viewed as a sequence of concurrent events. In the Java programming language the bytecode is nearly such a level. The reason it cannot be considered such a level is that operations on $longs$, etc., are not atomic.

the server thread might change its state while the code segment is executing. This would result in blocking the code segment. The lost notify bug pattern discussed in 2.3.2 is another example of categories that overlap.

In the following subsections we give examples of the aforementioned high level categories. We identify how bug patterns subset $I(P)$ and illustrate how these subsettings occur in practice. The examples given in subsections 2.2-2.4 are not complete. Further research is required to complete them.

2.2 Code Assumed to Be Protected

A code segment is protected, or undisturbed, for a concurrent program P, if for any interleaving in $I(P)$, and any execution of the code segment by some thread we use threads and processes interchangeably in this article in the interleaving, no other thread executes a concurrent event between the execution of the first concurrent event in the code segment and the execution of the last concurrent event in that segment.

A bug pattern occurs when a code segment is assumed to be protected but is actually not. In practice this bug pattern occurs in many flavors. We detail some of these flavors in the following subsections.

2.2.1 Nonatomic Operations Assumed to Be Atomic

Sometimes the bug pattern is related to the programming language abstraction level. For example, the Java $x++$ operation for a class instance field is often thought to be protected because it seems to consist of one operation. In fact, however, the bytecode translation of this operation consists of first moving the current value of x from the heap to the thread's local area copy of x, then incrementing the thread's local area value by one, and finally updating the heap value of x [3]. A context switch may occur after each of these stages, and as a result, the code is unprotected.

In this instance we sometimes say that the operation is nonatomic. The bug pattern consists in this case of 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. The programmer is only aware of one operation, $x++$ in our example, at the high abstraction level and wrongly assumes that this operation is processed "the way it looks," that is, as one operation.

2.2.2 Two-Stage Access Bug Pattern

Sometimes a sequence of operations needs to be protected but the programmer wrongly assumes that separately protecting each operation is enough. A concrete example follows. Adding and removing operations to some data base are performed concurrently by first accessing a table to translate from $key1$ to $key2$. Then $key2$ is used to access another table and add or remove the data. Accesses to both tables are synchronized. However, the code segment that accesses the two tables is left unprotected between the access to the first and the second table. This causes a concurrent bug because the programmer ignores the fact that, after a thread obtained $key2$ from the first table, a context switch may occur, and other threads may access and change the two tables.

[3]This description is correct at least until Java version 1.3.

2.2.3 Wrong Lock or No Lock

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. As a result, interference between other threads and the code segment to be protected are possible and the code segment is actually not protected.

Consider for example, a given global integer variable x with initial value 0. The first thread executes

$synchronized(o)\{x++\}$

while the second thread executes

$x++;$

A possible final value for x is also 1 because $synchronized(o)\{x++\}$ is actually not protected. Both threads may update the local area values of x to zero, increment it to 1 and update the heap value of x to 1, twice.

This bug pattern is actually more general. It might occur whenever some access protocol is required for accessing a system shared resource and the programmer does not follow the required protocol.

2.2.4 Double-checked Locking

Object initialization in concurrent programs is tricky. An example of this is a well-documented concurrent bug pattern called "double-checked locking."[4] When an object is initialized, the thread local copy of the object's 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. Again, the initialization operation is misleading at the source code level, and the possible interleaving of the actual assignments to the thread local memory area and the heap are not taken into consideration by the programmer.

[4]For details, see $http//www.javaworld.com/javaworld/jw02$-2001/$jw0209double\_p.html$.

2.3 Interleavings Assumed Never to Occur

In this bug pattern the programmer assumes that a certain interleaving never occurs because of the relative execution length of the different threads, or because of some assumptions on the underlying hardware, or some order of execution "forced" by explicit delays, such as $sleep()s$, introduced in the program. In other words, interleavings in $I(P) - C(P)$ are considered as impossible in practice, or not considered at all, by the programmer.

This bug pattern comes in various flavors. We detail some examples in the following subsections.

2.3.1 The sleep() Bug 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. The correct solution would be for the parent thread to use the $join()$ method to explicitly wait for the child thread.

2.3.2 Losing a Notify Bug Pattern

If a $notify()$ is executed before its corresponding $wait()$, the $notify()$ has no effect and is "lost." As a result, code executing a $wait()$ might not be awakened because it is waiting for a $notify()$ that occurred before the $wait()$ was executed. In this bug pattern, the programmer implicilty assumes that the $wait()$ operation will occur before any of the corresponding $notify()$ operations, i.e., that a certain interleaving will never occur. Again, this implicit assumption can be "supported" by adding appropriate delays. One way of avoiding this bug pattern is to repeatedly execute the $notify()$ operation until a condition stating that the $notify()$ was received occurs.

Note that this is an instance of categories that overlap. Although we could classify this bug pattern as the "interleavings assumed never to occur" category, we could also classify it as the "blocking or dead thread" category because the $wait()$ statement might block if the $notify()$ statement is lost.

2.4 Blocking or Dead Thread Bug Pattern

In this bug subcategory some interleavings in $I(P) - C(P)$ contain a blocking operation that blocks indefinitely. Alternatively, one of the threads terminates, typically by taking an exception, when it was not expected to terminate. Sometimes this bug manifests as a hang of the system. We give several examples below.

2.4.1 A "Blocking" Critical Section Bug Pattern

A thread is assumed to eventually return control but it never does. This situation may occur in a critical section protocol [10]. A thread entering the critical section is assumed to eventually exit it. If the thread executing within the critical section performs some blocking I/O operation (file, socket or GUI interface waiting for an event), it may never exit.

This bug pattern is sometimes manifested by a hang of the system, when all threads are waiting for a thread in the critical section to exit but the thread never does.

This bug pattern also appears when a server is receiving requirements for operations from a queue. The server $main()$ function repeatedly removes a requirement for an operation from the queue and then executes it by calling a service. If the service is called in the same thread, it is assumed that this service will eventually return. Again, for the reasons mentioned above, this service might never return, thus causing the server to hang.

Another variation of this bug pattern occurs when a server's service method, known to be non-blocking, is overwritten by the incorrect use of inheritance with a blocking method. In addition, service methods can be long or written by different project members (or by a third party, which is even worse from a testing perspective). Thus it is hard to guarantee that the service method is nonblocking.

2.4.2 The Orphaned Thread Bug Pattern

This bug pattern is documented by Eric E. Allen in his 2001 developerWorks article.

In multithreaded code, it is common to use a single, master thread that drives the actions of the other threads. This master thread may send messages, often by placing them on a queue, that are then processed by the other threads. If the master thread terminates abnormally, the remaining threads may continue to run, awaiting more input to the queue and causing the system to hang.

3 New Timing Heuristics for Finding Concurrent Bug Patterns

ConTest [4] timing heuristics are written to increase the probability that known kinds of concurrent bugs will occur. ConTest architecture is described in detail in previous works ([4] and others). For the purpose of describing timing heuristics it is enough to understand the following. At runtime ConTest is given control before and after a concurrent event is executed. At that time ConTest is passed various kinds of information such as the name of the statement being executed or a reference to the variable being accessed. In addition, sometimes ConTest is given control of whether to invoke the concurrent event at all or override it by a different, semantically equivalent operation. Thus, ConTest architecture can be viewed as an instance of the filter design pattern. At runtime, when control is passed to ConTest before and after a concurrent event is executed, ConTest uses Java primitives such as sleep() and yield() to change the order of concurrent event execution. It follows that a timing heuristic is a description of code segments executed before or after a concurrent event is encountered and a statement of whether the original concurrent event is executed.

The development of additional timing heuristics is an area for further research. Categorizing concurrent bug patterns, as we do in this article, can facilitate this research. In this section we give some concrete examples of how concurrent bug patterns can motivate the creation of new general purpose heuristics.

We attempt to change the order in which a $o.notify()$ and $o.wait()$ occur so that the chance that the "lost notify" concurrent bug pattern (section 2.3.2) will occur is increased. A concurrent event is chosen randomly and then, before the concurrent event is executed by the program under test, ConTest executes this protocol:

```
otherAdvancing = true; // first assume that other
                       //threads are advancing
while(otherAdvancing){
 otherAdvancing = false;
 sleep(duration); //wait for other threads to advance
}
```

Other threads, other than the randomly chosen thread, execute a different protocol before a concurrent event is executed:

```
otherAdvancing = true;
```

In this description we disregard atomicity issues. The description should be regarded as atomic whenever this is required for correctness. The heuristic identifies that no thread is likely to advance. This is done with minimal synchronization. The chosen thread advances when it discovers that no other thread is likely to advance.

Assume that the chosen thread is a thread about to execute an $o.wait()$ concurrent event. Further assume that this thread is executing the first code segment above just before it obtained the synchronized lock that encloses an $o.wait()$ concurrent event. In this case the chosen thread is likely to pause long enough to increase the chance that an $o.notify()$ operation will be executed and lost. Thus, it is intuitively clear that this timing heuristic increases the chance that the lost notify bug pattern manifests. Further experiments are required to validate this claim.

This heuristic has additional merits. It produces legal interleavings that are not usually produced by typical schedulers. It can thus be viewed as a boundary condition test. Intuitively, it increases the chance that code segments assumed to be undisturbed by the programmer are actually disturbed. Thus, it seems useful for finding all bug patterns in the "code assumed to be protected" category. Experiments are required to validate this statement.

This heuristic can be further developed. If we want to focus on the "lost notify" concurrent bug pattern, we must identify related $o.wait()$s and $o.notify()$s, i.e., $notify()$s and $wait()$s executed on the same reference. This can be done by using histories of previous runs or by using static analysis information. Only threads that have $o.wait()$s operations expected to have related $o.notify()$s operations are prevented from advancing.

Another general purpose heuristic that helps uncover the $sleep()$ bug pattern (2.3.1) is to randomly change, at run time, the $sleep()$ durations used in various places by the program under test. The choice of sleep duration is random within the original duration used at each $sleep()$ execution point of the unaltered program. Sometimes we do not call the $sleep()$ method at all. It is our experience that this heuristic is effective for early revealing of concurrent code that is nonportable due to implicit assumptions on the speeds of various hardware or software components.

4 Deducing Concurrent Bug Patterns from Design Patterns

Rather than testing for concurrent bug patterns, a tester can deduce effective concurrent tests from a given concurrent design pattern [7]. Concurrent design patterns can also be used to deduce typical concurrent bug patterns.

In this section we give several examples of how concurrent design patterns can be used to deduce some of the concurrent bug patterns discussed in the previous section. We hope that these examples will facilitate more thorough research on the relation of bug patterns and design patterns. A similar line of research can be taken with antipatterns.

A design pattern [5] describes a recurrent problem and the core of the solution to that problem. Solutions are described in terms of objects, interfaces, and their relations. The description is detailed enough so that typical bugs and useful general test techniques suggest themselves.

The descriptions of the concurrent design patterns in the following subsections are not complete. For a complete description, please refer to [7] and [8].

4.1 Confinement and Subclassing

Subclassing is used to access the bare class in confined mode in some contexts and in nonconfined mode in other contexts. A confined mode is a mode in which all methods are protected, usually by a lock, and in which “leak” of references to class instances other than the confined class is prevented. A typical defect pattern in this case is that the confined version is used in a context that requires the nonconfined version or vice versa. The former has a performance impact and the latter results in undesired interference.

It is interesting to note that the second version above falls under the category of the "code assumed to be protected" bug pattern and under the wrong lock or no lock bug pattern example (subsection 2.2.3).

4.2 The Token Design Pattern

A token is an exclusively held resource that is analogous to a physical object. Only one party can hold a token at any given time. If the token is destroyed, it cannot be used at all.

A typical bug pattern associated with the token design pattern is described below. Since the resource is not physical, transfer is not atomic. For example, after a "resource transfer" of the form $x.r = y.s$, object y still points to resource s. A proper transfer should look like $x.r = y.s;\ y = null;$.

This defect is associated with the $code\ assumed\ to\ be\ protected$ concurrent bug pattern 2.2. If $x.r = y.s;\ y = null;$ is protected then the bug will not occur.

4.3 The Fork/Join Design Pattern

The Fork/Join design pattern subsets a problem into subproblems and solves them in parallel. For example, the following code segment could be used.

```
IN − PARALLEL{
  h = solve(half(problem));
  o = solve(theOtherHalf(problem));
}
 combine(h, o);
```

The Fork/Join Defect Pattern has several typical bugs. For example, nonvolatile variables are sometimes used. As a result, the updated value is not viewed at the combine stage. Another problem is that the task waiting for the ‘final result’ attempts to obtain the ‘final result’ before computations have ended.

The first of the two aforementioned bugs is related to the concurrent bug pattern discussed in subsection 2.2.1, and the second is related to the concurrent bug pattern discussed in section 2.3.1.

5 Conclusion

In this paper we describe and categorize a taxonomy of concurrent bug patterns. In addition, we indicate how to use that new taxonomy to enhance ConTest’s [4] ability to find concurrent bugs. Although industrial experience shows positive results, formal experiments are needed to validate them.

Most of the taxonomy is not programming language specific: it is related to the synchronization primitives used in Java, but similar synchronization primitives are used in other environments, such as pthread [9]. In addition, instances of some of the bug patterns discussed in this paper appear in other environments in slightly different forms. For example, the lost notify bug pattern also occurs in the Unix environment as a “losing single bug pattern”: signals are lost when they are sent before the signal handler is established. This example suggests that an additional layer of abstraction in the taxonomy description could be introduced.

Other testing techniques can be enhanced and developed using the taxonomy. For example, new coverage models can be introduced. A concrete example is the following coverage model aimed at testing for the lost notify bug pattern (see subsection 2.3.2). For each $notify()$ operation that may occur concurrently to a $wait()$ operation, create two coverage tasks: one for the $notify()$ before the $wait()$ and the other for the $wait()$ before the $notify()$.

To facilitate research in testing of concurrent programs it is useful to further develop the taxonomy. We demonstrated how concurrent design patterns can be used to deduce concurrent bug patterns. Further research is required to thoroughly understand the relation of bug patterns and design patterns. A similar line of research can be taken with antipatterns.

The introduction of bug taxonomies facilitated the search for new sequential program testing techniques. We hope that the concurrent bug pattern taxonomy will serve the same purpose for concurrent programs.

References

[1] B. Beizer. *Software Testing Techniques, Chapter Two*. International Thomson Computer Press, 1990.

[2] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*. Wiley, 1988.

[3] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[4] E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Java Grande/ISCOPE 2001 Special Issue of Concurrency and Computation: Practice and Experience*, 2001.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques, Chapter Seven*. Morgan Kaufmann, 1993.

[7] D. Lea. *Concurrent Programming in Java, Design Principles and Patterns, First Edition*. Addison-Wesley, 1999.

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

[9] B. Lewis and D. J. Berg. *A Guide to Multithreaded Programming. Appendix E.* SunSoft Press. A Prentice Hall Title, 1996.

[10] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[11] S. D. Stoller. Testing concurrent java programs using randomized scheduling. *In Proceedings of the Second Workshop on Runtime Verification (RV)*, 2002.