A Classification of Concurrency Bugs in Java Benchmarks by Developer Intent

M. Erkan Keremoglu
Koc University
College of Engineering
Istanbul, Turkey
mkeremoglu@ku.edu.tr

Serdar Tasiran
Koc University
College of Engineering
Istanbul, Turkey
stasiran@ku.edu.tr

Tayfun Elmas
Koc University
College of Engineering
Istanbul, Turkey
telmas@ku.edu.tr

ABSTRACT
This poster discusses concurrency bugs that result from violations of race-freedom, atomicity and refinement criteria around real examples and emphasize the importance of taking into consideration each criterion. Our purpose is to demonstrate the fact that the proper fix for a concurrency error and the proper program analysis tool for detecting such errors must be based on a formalization of the designer’s intent by designating what high-level operations must run atomically.

1. INTRODUCTION
Concurrent software is increasingly more common. This makes functional errors due to concurrency a central issue. Concurrency errors can be very difficult to catch and may have serious consequences [7].

The usual solutions to avoiding concurrency errors in Java programs involve using synchronization mechanisms such as monitor-based locking. For large-scale applications, avoiding concurrency errors and achieving high performance are conflicting forces. Protecting large pieces of code using locks often has an unacceptable impact on performance. Similarly, protecting too small a block of code or too few objects with locks often leads to subtle concurrency errors.

For these reasons, making precise the developer’s intent and identifying the corresponding correctness criterion is of key importance. The most appropriate fix for a concurrency error can only be determined based on this information. Further, the program analysis tool that is best suited for a system is one that takes the appropriate correctness criterion as its basis. To demonstrate these facts empirically, on a set of bugs in Java benchmark programs reported in the literature, we contrast three correctness criteria for concurrent software: race-freedom [2, 1], atomicity [6, 5] and refinement [4]. Our findings illustrate that concentrating verification tool and developer effort on bug symptoms rather than the violated intent is the wrong approach.

We start with bugs detected by various tools in real programs [1, 4, 6, 5] and investigate what behavior of the program is intended by the developer and which correctness criteria best expresses it and leads to the proper bug fix.

For each concurrency error, we investigate the following aspects:

- **Bug manifestation**: This is the observed discrepancy in program behavior caused by the bug. Examples are null pointer dereferences and data corruption. While some of these errors can be detected upon their occurrence by the runtime system, others, such as data corruption, may go undetected.

- **Symptoms detected by tools**: Program analysis tools that target concurrency errors have as their focus a category of symptoms, such as race conditions or code patterns frequently associated with concurrency bugs. The association between bugs and these symptoms is often indirect. The symptoms may be neither the conceptual source of the error nor its manifestation. Frequently, modifying code by focusing on removing a symptom causes other bugs and/or symptoms.

- **Appropriate correctness criterion and bug fix**: The correctness criterion formalizes the properties that must be satisfied by program so that they execute without unintended side effects of concurrency. For each program, we consider race-freedom, atomicity and refinement as possibilities for the correctness criterion that best expresses the developer’s intent.

Race-freedom ensures a total order on operations on data variables throughout the execution, and allows the programmers to reason about the program with sequential consistency semantics [3]. Typically, however, the programmer has a higher-level correctness goal in mind for which race-freedom may not be sufficient, or, in some cases, necessary. A more appropriate correctness criterion for such programs may be atomicity. Atomicity of a code block ensures that the program variables accessed by the block act as if there were no interference by the environment. While in some designs it is the appropriate goal, making large code blocks atomic leads to an unacceptable performance hit in terms of the degree of concurrent operations. A more permissive correctness criterion, refinement, requires that only an abstract view of the program state and not the whole state itself be updated atomically. The use of refinement as the correct-
ness criterion often leads to smaller synchronized blocks and a higher degree of concurrency. We refer to a code block that properly refines its specification as "abstract-atomic".

In the absence of annotations that indicate the developer's intent, such as atomicity or refinement annotations for code blocks, tools are forced to focus on low-level symptoms such as races. We believe this both makes it laborious to identify the root cause of and a fix for the bug, even worse, it misleads the developer to treating the symptom rather than the cause. It is one of our goals in this study to empirically demonstrate this claim. The following section elaborates our considerations around a real example.

2. EXAMPLES

The example explained in this section is from one of the benchmarks used by Naik et. al. in [1]. We have chosen this example because fixing the bugs in the example is, judging by the discussions between the developers and Naik, not trivial and aims to strike a compromise between performance and correctness criteria that express the intent of the developer. Although the symptoms of the bug consists of a set of race conditions, avoiding the races does not suffice to ensure the intended behavior, but a more elaborate synchronization policy is needed. We proposed a fix for the bug, following the considerations we want to emphasize in this paper.

The example involves the RequestHandler class of Apache FTP Server, which is a free, open-source FTP server. Upon a new connection being established between a client and the server, a thread is started in order to handle requests on this connection by using an instance of RequestHandler. The corresponding run() method of this connection is shown in Figure 1. This method has a loop body which listens to incoming FTP requests from the client. In the body of the run() method, fields m_request, m_writer, m_reader and m_controlSocket are accessed.

The close() method is shown in Figure 2. When close() is called, if the connection is still open, it sets the m_isConnectionClosed field of the current RequestHandler instance to true and sets the m_request, m_writer, m_reader and m_controlSocket fields to null. run() and close() are designed to be run concurrently by different threads. The current synchronization policy allows close() to close the connection while the loop body in run() is in progress. As a result, suppose that run() checks m_isConnectionClosed at line 22, detects that the connection is still alive. Before it tries to read a new request at line 7, a context switch occurs and another thread calls close() on a request handling code (lines 3 and 4) before the request handling code gets into the request handling loop (lines 6-22, inclusive). The fix can be seen at lines 2-4 of the fixed code in Figure 3.

The next solution that naturally comes to mind is to make run() and close() entirely synchronized on this. This solution is not acceptable, however, since making run() and close() run mutually-exclusively disables another thread from closing a connection by running close() since once started run() does not terminate unless m_isConnectionClosed is set to true.

The developer, though, has to allow another thread to call close() concurrently by reducing the granularity of synchronization between run() and close(). The entire body of close() can be made synchronized since it is called just once and no other thread should be allowed to run after it is called. But the developer must solve this problem by implementing run() using more than one atomic block.

First observe that there is some connection initialization code (lines 3 and 4) before the request handling code gets into the loop at line 6. Synchronizing initialization block will not dramatically decrease the performance of the program since it is called only once when a new connection arrives. In addition it is assumed that initialization takes small time and it is low probability that a connection is closed before it is initialized. Thus, the fix will first include moving the whole initialization step into a method called init() synchronized on this. The fix can be seen at lines 2-4 of the fixed code in Figure 3.

Making synchronized the entire request handling block (the loop in lines 6-22, inclusive) does not eliminate the concurrency error and is unacceptable for other reasons as explained next. The first reason is that another thread may run close() before line 7 of run() is executed and set m_request to null. This would cause run() to throw a null pointer exception at line 7. Another reason is that if run() gets into the request handling loop while the connection is open, another thread can not run close() to set m_isConnectionClosed due to the synchronization on this. This prevents run() to terminate as it always sees m_isConnectionClosed false.

The next natural attempt at fixing the bug would be to reduce the synchronization granularity by making each iteration of the request handling loop (lines 7-21, inclusive) of run() synchronized on this separately. This allows another thread to close the connection between two iterations of the loop handling two requests.

With this modification, the null pointer dereferencing problem still persists inside the body of the while loop even if the atomicity of handling each request is ensured. If a thread running close() sets m_reader to null before run() starts handling a new request (at line 7) but after checking m_isConnectionClosed at line 22, dereferencing m_reader at line 7 for the next request causes a null pointer exception.

For a proper fix, the designer's intent must be formalized by making explicit what operation needs to be performed wrapping each access to the fields m_request, m_writer, m_reader, m_controlSocket in a separate synchronized block on this, and declaring m_isConnectionClosed as volatile. Even though we eliminated all the races in the code, the bug described above has not been disappeared yet. It is still possible for close() to run concurrently between run()'s checking the m_isConnectionClosed field and dereferencing m_request, m_writer, m_reader in the loop body.
atomically. For correct operation, `m_isConnectionClosed` must be checked before execution of each iteration in the same atomic block. One can allow more concurrency and preserve correctness by observing that the loop body performs two high-level operations: (i) reading and parsing the command and (ii) servicing the request in the command. Breaking the loop body into two code blocks and making each block atomic with a check that `m_isConnectionClosed` is `false` as shown in Figure 3. This both fixes the concurrency error and allows the highest degree of concurrency by allowing the connection to be closed in the middle of the loop. Note that breaking the loop body into two atomic blocks is dependent on the intent of the developer whether she wants to allow or prevent connections to be closed while there is a request already read by `m_reader`.

```java
1 public void run() {
2 ...  
3 dereference m_request, m_writer, m_reader
4 and m_controlSocket to initialize the connection
5 ...  
6 do {
7 String commandLine = m_reader.readLine();
8 if(commandLine == null) {
9 break;
10 }  
11 commandLine = commandLine.trim();
12 if(commandLine.equals("")) {
13 continue;
14 }
15 m_request.parse(commandLine);
16 if(!hasPermission()) {
17 m_writer.send(530, "permission", null);
18 continue;
19 }
20 // execute command
21 service(m_request, m_writer);
22 } while(!m_isConnectionClosed);
...  
```

Figure 1: The buggy code fragment

```java
1 public void close() {
2 // check whether already closed or not
3 synchronized(this) {
4 if(m_isConnectionClosed)
5 return;
6 m_isConnectionClosed = true;
7 }
8 ...  
9 m_request = null;
10 ...  
11 m_writer = null;
12 ...  
13 m_reader = null;
14 ...  
15 m_controlSocket = null;
...  
```

Figure 2: The close method

3. CONCLUSIONS

The purpose of the example in Section 2, and others we will present in this poster, is to demonstrate the fact that the proper fix for a concurrency error and the proper program analysis tool for detecting such errors must be based on a formalization of the designer’s intent by designating what high-level operations must be made atomic or abstract-atomic.

```java
1 declare m_isConnectionClosed field volatile
2 ...
3 synchronized(this) {
4 init();
5 }
6 do {
7 synchronized(this) {
8 if(m_isConnectionClosed)
9 break;
10 else
11 String commandLine = m_reader.readLine();
12 }
13 if(commandLine == null) {
14 break;
15 }
16 commandLine = commandLine.trim();
17 if(commandLine.equals("")) {
18 continue;
19 }
20 synchronized(this) {
21 if(m_isConnectionClosed)
22 break;
23 else{
24 m_request.parse(commandLine);
25 if(!hasPermission()) {
26 m_writer.send(530, "permission", null);
27 continue;
28 }
29 // execute command
30 service(m_request, m_writer);
31 }
32 }
33 } while(!m_isConnectionClosed);
...  
```

Figure 3: The fix for the bug

4. REFERENCES


