A Methodology to Detect "Hard-to-Find" bugs in Large Multithreaded Java Programs

Sreeranga P. Rajan and
Thomas Sidle
Fujitsu Laboratories (FLA)
Sunnyvale, CA
sree.rajan@us.fujitsu.com

Graham Hughes
University of California
Santa Barbara, CA

Keith Swenson
Fujitsu Software
Corporation
Sunnyvale, CA

ABSTRACT

Multithreaded programming has become increasingly essential for developing highly efficient software. Concurrency in multithreaded programs introduces additional complexity in software verification and testing, and thereby significantly increasing the cost of Quality Assurance (QA). In this report we describe the verification of enterprise client-server Business Process Management (BPM)/workflow software. We developed a practical verification methodology using a formal analysis tool called Java PathFinder (JPF) from NASA with special emphasis on finding deadlocks and race conditions. The results revealed race conditions that lead to data corruption errors whose detection would have been prohibitively expensive with conventional testing and QA methods. To the best of our knowledge, this is the first time a methodology involving formal analysis has been successfully applied to such a large client-server software project.

1. BACKGROUND AND HISTORY

Sree Rajan and Tom Sidle of Advanced CAD Department, Fujitsu Laboratories of America (FLA), initiated the contact with Keith Swenson, Chief Scientist and Architect of Interstage Business Process Management (I-BPM) and Director of Fujitsu Software Corporation in 2003. In this meeting we concluded that there was scope for applying formal verification techniques to detect deadlocks and race conditions in Fujitsu's workflow client-server Java software. We established research links with Willem Visser of NASA and obtained the model checking tool called Java Path Finder (JPF) for trial purposes. We then had several meetings with the software development team and did preliminary trials on applying JPF to the workflow software. As it is well known that applying formal model checking tools to large software or hardware designs fail, we came up with a strategy to apply model checking only to portions of the large workflow software with promising results. In July 2004, we attended the Computer-Aided Verification (CAV) conference and met with Willem Visser. With his acceptance to collaborate in the software verification of a real large software product, NASA agreed to support our software verification activity in providing JPF and also funding for a student intern if requested. After a month-long search for a competent doctoral student researcher, Graham Hughes from UCSB accepted to visit us starting August 23, 2004 for a few months to work on verifying I-BPM software using JPF. Initially, the most difficult part was how to select portions of the I-BPM code on which we need to perform model checking. I-BPM is a complex and mature product, with more than 1500 classes spanning more than 500,000 lines of Java code. We were challenged to discover deadlocks and race conditions in the cache and other portions of I-BPM that they could not find using conventional tools. We met this challenge successfully in 5 weeks by the end of September 2004. Our methodology has helped increase the quality of the product in the field, thereby demonstrating the effectiveness of formal analysis in detecting complex concurrency bugs. All the bugs reported in this memo have or had been fixed in the field.

** This is the first time a successful attempt has been made in applying formal analysis for finding bugs in very large client-server industrial strength software. Interstage Business Process Management (I-BPM) software developed and marketed by Fujitsu Software Corporation is a complex and mature product, with more than 1500 classes spanning more than 500,000 lines of Java code. The challenge in finding bugs in such a large software program is as hard as finding a small needle in big haystack. Our methodology has helped increase the quality of the product in the field. All the bugs reported in this memo have or had been fixed in the field.

2. INTRODUCTION

Nondeterminacy is the possibility that a concurrent program yields different outputs for different runs with the same input data [RAM85]. Errors such as deadlocks, livelocks and race conditions are latent in concurrent shared-memory programs. Without a formal proof, the very existence of these errors can be very hard to find.

A *data race* condition exists when multiple entities (threads/processes) concurrently read and write the same data, and the outcome of the execution depends on the particular order in which the accesses take place. In general, detecting race conditions is NP-hard. *Deadlock* occurs when a multi-threaded program is unable to make progress because a thread is waiting for a condition that will never happen.

Deadlocks and race conditions are the two most significant problems that occur in concurrent programs. Java supports concurrent programming that includes synchronization primitives "wait" and "notify" for controlling access to shared resources by multiple threads.

In this project we considered the Java-based flagship Business Process Management (BPM) product, named Interstage Business Process Manager (I-BPM), which is developed and marketed by Fujitsu Software Corporation.

We have developed a methodology to detect correctness problems due to concurrency associated with large multithreaded programs. We use a Java program model checker called Java PathFinder (JPF) [2] developed at NASA. It implements a Java virtual machine permitting direct formal analysis of Java programs, unlike many other formal analysis tools. We also provide a set of recommended programming discipline to minimize correctness errors in multithreaded

Java programs.

3. I-BPM OVERVIEW

The system [1] has a 4 tier architecture as shown in Figure 1. The user interface can be implemented either as a Java based thick client, Java applets, or as web pages in a browser. The browser is served by the web tier, which is composed from JSP and servlet components running in a web server. The main process logic and the analytics engine reside in the BPM tier. The fourth tier contains the underlying repositories such as database, directory, and document management as well as connectivity to other systems using a variety of mechanisms. Various modules that consitute the interfaces between different tiers use Java RMI.

4. VERIFICATION METHODOLOGY

I-BPM has been in existence in the field for a number of years, and therefore our exercise in detecting bugs in this product is akin to finding a needle in a haystack. Applying formal analysis tools to large software programs has proved to be a futile exercise. Practical multithreaded programs typically are plagued by a large state space that is beyond the capabilities of formal analysis tools. Adding to this difficulty are the File and Network I/O and library constructs that are not within the scope of JPF and other model checkers. Together these reasons compel us to provide a verification methodology that involves formal analysis tools such as JPF.

Our verification methodology comprises of 5 *repetitive* steps:

- Bring to focus possible ”buggy regions” by rapid high-level analysis and a description of analomous behavior from the software team;
- Preparing the environment for applying formal analysis, including stubbing out RMI and I/O calls;
- Abstracting out irrelevant details—for example, references to localization as part of java.locale were irrelevant to the goals of this project;
- Applying JPF;
- Analyzing the results from application of JPF

We were informed that the product in the field was susceptible to deadlocks, the frequency of which went up when different databases were used. This suggested to us that the subsystem associated with the database would be a good place to start.

Because I-BPM is broken up into subsystems that use RMI to communicate, the network protocol that they use to communicate with the database subsystem is exposed. The relevant part of this RMI protocol shown in Figure 2 is as follows:

1. Client code acquires the subsystem’s published interface, `DbAdapter`, using RMI.
2. Client code calls `getConnection()` on the `DbAdapter`, getting a `DbConnection`.
3. Client code uses the database through the `DbConnection`.
4. Client code releases the connection; return to step 2.

This turns out to be too simple for real use. The database subsystem attempts to avoid creating and destroying `DbConnections` indiscriminately, on the grounds that this is very slow. So the release in step 4 actually returns that `DbConnection` to a pool that connections are selected from in step 2. Now we have a different problem; if a client dies for some reason while it holds a DbConnection, or there is some network trouble, or if for any other reason a client never gets around to performing step 4, that DbConnection will never be returned to the pool and a resource leak will be created. To solve this problem, I-BPM checked each of its connections in step 2 to see if the original client is still alive, and will hand out an already allocated connection to the new client, believing that the original client is dead.

By itself, this is harmless; however, I-BPM’s method of determining whether or not a client is dead is whether the connection is five minutes old. If for any reason—database row locks, database backup, network latency, heavy load, just taking a while—a client takes more than five minutes to process a transaction, it runs the risk that its connection will be given to some other client.

This is a disaster waiting to happen; the two clients transactions will be interleaved, and additionally the JDBC standard does not mandate that `java.sql.Connection` objects be thread safe.

5. CACHE VERIFICATION

After analysing this protocol, at the request of the development team we began analyzing part of the object cache, whose goal is to make accesses to the database no more frequent than is totally necessary. We specifically concentrated on the `ProcessDefinitionProxy` class, which acts as a layer between the main code and the database adapter specifically focused on `ProcessDefinitionImpl` structures.

There were a number of concurrency errors present; a list of recommendations for avoiding them is presented in Section 6. Many of these errors were harmless because of the use of `java.util.Hashtable`, which does its own locking, but many were not.

There were several public methods that accessed shared data without locking it appropriately; these methods were:

- `saveBrandNew` (which is harmless because of `Hashtable` locks
- `comboCommit` (which causes a race between two remove invocations)
- `comboEdit`
- `edit` (which is harmless because of `Hashtable` locks
- `commit` (which causes a race between get and put but only if the id is shared)
- `destroy` (which causes a race between two remove invocations)
- `fetchFromServer` (not public but insufficient locking discipline causes public methods that call it to be faulty)
- `invalidateCache` (which causes a race between a get and a remove)

We also discovered several public methods that accessed shared data without any synchronization whatsoever; these were:

- `setWFSession`
- `setLastSaved`
- `addPlanListener`
- `removePlanListener`
- `fireEvent`
- `isPlanChanged`
- `setTriggerState`

The object caches were publically visible and several other classes used them; the class `WFAdminSessionImpl` goes so far as to modify the caches directly, again with no locking discipline at all.

Some miscellaneous synchronization oddities discovered were

- `getUniqueId` superfluously synchronizes on a static object after synchronizing on the class,
- `getNewId` synchronizes on a per-object lock, which accomplishes nothing.

An additional serendipitous discovery was an initialization error in the `ProcessDefinitionProxy` object; when a `ProcessDefinitionProxy` reads a process definition from the database, it caches the identifier in the `pdid` field, a number uniquely determining the process. When a new process definition is created, this identifier has not yet been determined; after the process definition is saved to the database, this identifier cache should be set to the actual value instead of the default value of 0. This necessary update was omitted.

This bug was discovered during a stress run through the system using JPF; we had no idea that it existed and just wanted to verify that the system behaves appropriately in the presence of several threads working on different process definitions. We discovered that they were all attempting to work on the same database object (that being object #0) and spent some hours trying to determine how we were "misusing" the system to cause this to occur, before we traced it to this cache update never occuring.

6. RECOMMENDED PROGRAMMING DISCIPLINE

There is a marked difference in kind between the errors described in Section 4 and those described in Section 5; the former exposes a subtle flaw in the protocol two subsystems use to communicate, whereas the errors in the latter section largely come from insufficient or incorrect locking discipline. A more rigid coding style can pick up the "low hanging fruit" of insufficient locking discipline, although it will not have any real effect on larger architectural flaws.

A comment, first: locking discipline is only important if a Java object is to be used in multiple threads simultaneously. Many Java objects are written under the assumption that this will never occur. However, because the computation model for threads exposes the entirety of local storage to concurrent modification, it is very easy for one insufficiently careful piece of concurrent code to violate this assumption.

One cure for this is to carefully quarantine the business of concurrency to a few gatekeeper objects, and this is the architectural style that I-BPM used. There are a great many objects in a Java system that become gatekeepers by default; anything that implements a remote interface exposes itself to the Java RMI implicit concurrency (wherein multiple consumers of a published interface are mirrored in the multiple independent threads of control accessing that same interface, none of which is made obvious through the use of RMI).

For this to work, there are several requirements of a gatekeeper:

1. Every method that accesses shared data must be protected by locks.
2. Every piece of shared data must be protected by the same lock each time.
3. Every method that returns data and every piece of public data must either return a newly created independent object for each client, or must return a gatekeeper object.

An interesting side effect is any object that is entirely read only (that is, the fields are set up by the constructor and never modified after the object has been created) is a gatekeeper even in the absence of locks, because the information can never change.

The I-BPM code base violated, largely by accident, omission or misunderstanding, each of these rules. Some of these turn out to be harmless because of the aforementioned `Hashtable` locking, but we must recommend against trusting the locking discipline of the entire enterprise to this, for two reasons:

1. the Hashtable locks are too low level; that is, while the Hashtable will prevent concurrent modification it does not protect you over the very frequent get/modify/return or isPresent/insert or isPresent/delete cycles;
2. serious concurrency errors can occur that do not involve the Hashtable but go unnoticed because the developers are used to leaving concurrency discipline up to the standard library.

6.1 Rule #1

Java contains built in constructs for thread locks in the `synchronized` keyword; an entire method can be marked as synchronized, or just a small block; in the latter case, the object to be locked must be specified explicitly; in the former it is the object the method belongs to (or the class object, if the method is static).

Every piece of shared data must be protected by a synchronized block, or by a synchronized method tag. An example correct usage, from the I-BPM source code:

```
public synchronized ProcessDefinitionProxy
    saveBrandNew ()
{
  ...
  if (!brandNew) {
    ...
  }
}
```

An example of incorrect usage, where the shared data is not locked:

```
public void setWFSession (WFSessionImpl wfs)
{
  ...
  if (brandNew && userAgentProxy == null) {
    ...
  }
}
```

And a fixed version:

```
public synchronized void
    setWFSession (WFSessionImpl wfs)
{
  ...
  if (brandNew && userAgentProxy == null) {
    ...
  }
}
```

Note that these tags are largely unnecessary for private methods if it is the case that every public method is protected by a lock; after all, there is no way that the private methods could be called without the lock. The same may be true of protected and default visibility methods, but because they are also accessible by objects in the same package as the class in question, more attention must be paid to them.

6.2 Rule #2

For the locks to be effective, the shared data must always be protected by the same lock. For object local data, this can be achieved by using the synchronized tag on the method; this becomes much more complex when an object is shared between objects.

I-BPM contained several cache bugs where methods that lock an instance of `ProcessDefinitionProxy` access static data that is a part of the class `ProcessDefinitionProxy`; as a result, when two objects try to use this shared data, they each lock different objects (themselves) and so the shared data is unprotected.

An example of incorrect usage (here `pdStructShare` is a static cache shared by all `ProcessDefinitionProxy` objects):

```
public synchronized void commit (String notUsed)
{
  ...
  ObjectCacheHolder hldr =
    (ObjectCacheHolder) pdStructShare.get (idkey);
  if (hdlr != null)
    sharedPdStruct = hldr.getProcDefStruct ();
  if (sharedPdStruct != null) {
    ...
    pdStructShare.put (idkey, ...);
  }
  ...
}
```

And now a fixed version:

```
public synchronized void commit (String notUsed)
{
  ...
  synchronized (ProcessDefinitionProxy.class) {
    ObjectCacheHolder hldr =
      (ObjectCacheHolder) pdStructShare.get (idkey);
    if (hdlr != null)
      sharedPdStruct = hldr.getProcDefStruct ();
    if (sharedPdStruct != null) {
      ...
      pdStructShare.put (idkey, ...);
    }
  }
  ...
}
```

6.3 Rule #3

The idea behind this is to preserve the gatekeeper properties; if a gatekeeper object somehow yields an object that is shared between threads that does not satisfy the gatekeeper properties, then trouble is inevitable.

I-BPM went out-of-the-way to do this, packaging up all model data in structures that are allocated freely per object. However, the object caches `pdCache` and `pdStructShare` are public objects, and were never locked properly; some of the classes that used this (for example, `WFAdminSessionImpl`) modified these caches directly, but more common was to use them to iterate over all available proxies and remove specified ones.

A better solution would be to move this functionality to the `ProcessDefinitionProxy` class, and then protect it behind a synchronized block. This would preserve the gatekeeper properties, and then the caches could be made private to the class. We became aware that the process of rearchitecting this portion of I-BPM is underway.

7. ARCHITECTURAL RECOMMENDATIONS

We recommend rethinking the protocol cited in Section 4 with an eye toward reliable disconnection; either the accessing object must be known to the database (so that its liveness can be reliably determined) or there must be a 1:1 relationship between remote accessors and server objects, and the pooling must be done on the database adapter side (so that disconnection can be reliably accomplished) or disconnection of dead threads must be abandoned.

This last option is easily dismissed as impractical in a real world environment, so one of the other two must be taken; the team has been reconsidering this part of the protocol for unrelated reasons and has decided to take option #2. We also recommend revisiting all objects that must deal with concurrency, and making them into gatekeeper objects, as recommended in Section 6.

8. DISCUSSION AND CONCLUSIONS

Deadlocks and race conditions are two sides of the "concurrency coin" that arise due to sharing violations in multithreaded programs. We chose to use Java programming language as an object of our effort due the absence of pointers that has been a constant challenge for formal analysis of imperative programs. This allowed us to focus our efforts on concurrency correctness problems rather than pointer analysis. We were also fortunate that we did not encounter state-space explosion, which would manifest in explicit-state model checkers as processing time unlike the problem of running out of memory in BDD-based model checking. Furthermore, our work attests to the common folklore in the verification and test community that if the system under validation has bugs, it will manifest not too deep in the state space.

9. SUMMARY AND FUTURE DIRECTIONS

In this report we have shown that formal analysis accompanied with a systematic methodology can successfully detect hard to find bugs in multithreaded programs. We detected two data corruption errors and a number of race conditions in Fujitsu I-BPM product. We have provided a number of recommendations advocating a discipline in multithreaded programming to avoid errors such as deadlocks and race conditions arising due to concurrency. All the bugs reported in this memo have or had been fixed in the field. To the best of our knowledge, this is the first time a methodology involving formal analysis has been successfully applied to such a large software project. As part of future work, we plan to capture the methodology as part of an Integrated Development Environment (IDE) for Java software development so that software developers at Fujitsu could use integrated formal analysis to detect hard-to-find bugs at an early stage.

10. ACKNOWLEDGEMENTS

This work may not have even begun without the existence of JPF, which was developed and kindly provided by Dr. Willem Visser of NASA. Dr. Visser backed it up with ”round-the-clock” guidance and support, without which we would not have succeeded. Shamim Quadir and Ivar Alexander of Fujitsu Software Corporation provided great help in understanding the software architecture and pointers to potential deadlocks and race conditions. This work would not have been possible without a strong suppport from Dr. N. Kawato. Dr. K. Kawamura and Dr. T. Nakata of FLL provided strong encouragement for this work.

11. REFERENCES

[1] Keith Swenson *Interstage iFlow Architecture White Paper*, Fujitsu Software Corporation, July 21, 2003.

[2] G. Brat, K. Havelund, S. Park, and W. Visser *I Java PathFinder - A second generation of a Java model checker* , Proceedings of Workshop on Advances in Verification, Chicago, Illinois, July 2000.

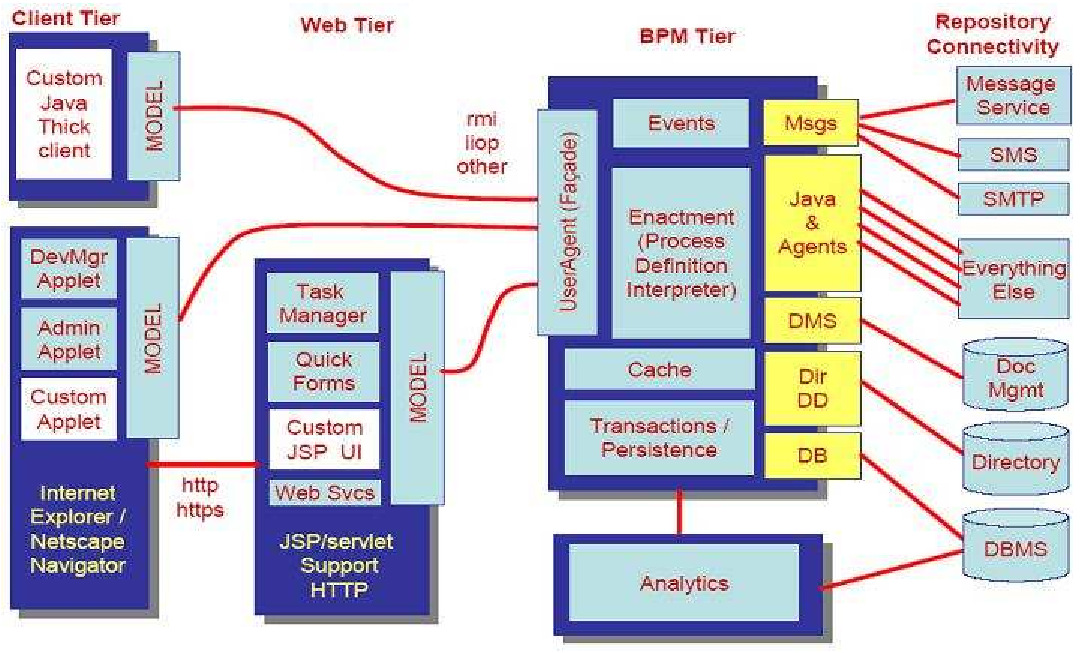

Figure 1: I-BPM Architecture

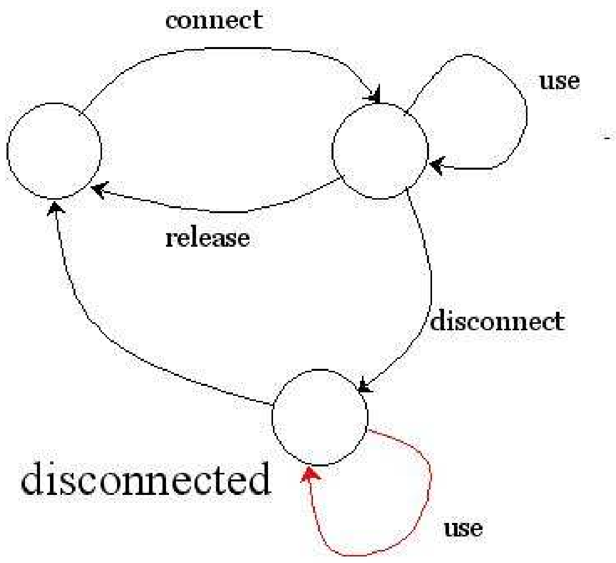

Figure 2: DBFSM Architecture