Model Checking Concurrent Red-Black Trees

Franck van Breugel
Department of Computer Science and Engineering, York University
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3

March 24, 2010

Abstract

In the previous assignments, we presented three concurrent implementations of red-black trees and discussed their implementations in Java. In this assignment, we discuss our model checking effort of the Java implementations using Java PathFinder.

1 Introduction

In [1], we presented three different ways to implement red-black trees concurrently. In [2], we presented their implementations in Java. In this third and final assignment, we model check some properties of our Java implementations. For the purpose we use the model checker Java PathFinder [4].

First of all, we use Java PathFinder to confirm that multiple threads manipulating a red-black tree concurrently using the operations Contains and Add without any means of synchronization leads to counter-intuitive results. Secondly, Java PathFinder does not detect any deadlocks for several simple tests containing multiple threads manipulating the tree concurrently. Thirdly, we show that those tests do not give rise to any uncaught exceptions. Fourthly, the absence of any data races in those tests is shown using Java PathFinder. And finally, we add some assertions to our Java code and show that these are not violated in any of the tests.

At least paragraph to discuss the remainder of the assignment is needed here.

2 Java PathFinder

Java Pathfinder (JPF) is an explicit state model checker for Java bytecode. It can check for properties such as uncaught exceptions and deadlocks. JPF has been developed at NASA. The development of JPF started in 1999. The first version of JPF was simply a translator from Java to Promela, the input language of the Spin model checker [3]. Since some features of Java cannot be easily represented in Promela, JPF was refactored as a Java virtual machine that can handle all bytecode instructions. The development of JPF became an open-source project in April 2005.

Recall that concurrent programs give rise to nondeterminism since the threads can be interleaved in different ways. Hence, a concurrent program may give rise to different executions. An ordinary Java virtual machine only considers a single execution. In contrast, JPF considers all possible executions in a systematic way.

While exploring all possible executions, JPF checks by default for properties such as the absence of deadlock and uncaught exceptions. Numerous extensions of JPF have been developed. For example, one can also check for the absence of data races. Furthermore, one can extend JPF to check for other properties as well. For more details about JPF, we refer the reader to, for example, [4][1]

3 Some Synchronization is Needed

In [1], we conjectured that multiple threads manipulating a red-black tree concurrently using the operations Contains and Add may lead to counter-intuitive results. Consider the following concurrent program.

```
1 Add(3)
2 Add(1)
3 (Add(2) || Contains(1))
```

We conjectured that by interleaving the elementary operations of the operations Add and Contains in a particular way, the operation Contains may return false. In [2], we reported that our Java implementation did not give rise to the anticipated counter-intuitive results.

The invocation Contains(1) is implemented in Java by a Thread. The run method of this Thread consists of

```
1 assert tree.contains(1);
```

Note that the assert failing (that is, throwing AssertionError) represents a counter-intuitive result. When we run Java PathFinder on the Java code, we obtain the following result.

```
% jpf +classpath=. sequential.Test
JavaPathfinder v5.x - (C) RIACS/NASA Ames Research Center

====================================================== system under test
application: sequential/Test.java

====================================================== search started: 3/14/10 7:46 PM

====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.AssertionError
        at sequential.Checker.run(Checker.java:35)

====================================================== snapshot #1
no live threads

====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
"java.lang.AssertionError at sequential.Checker.ru..."
```

[1] See also `http://babelfish.arc.nasa.gov/trac/jpf/wiki`.

```
====================================================== statistics
elapsed time:       0:00:01
states:             new=131, visited=50, backtracked=114, end=14
search:             maxDepth=69, constraints=0
choice generators:  thread=130, data=0
heap:               gc=204, new=593, free=312
instructions:       12968
max memory:         16MB
loaded code:        classes=82, methods=1071

====================================================== search finished: 3/14/10 7:46 PM
```

Hence, in one second Java PathFinder confirms that some sort of synchronization is needed to avoid counter-intuitive results.

4 Tests

Since JPF considers all possible executions, we have to keep our tests fairly simple in order to prevent JPF from running out of memory and to ensure that JPF terminates within a reasonable amount of time.

We started with a number of tests that only add integers to the tree. The first part of the test is sequential. In this part the integers $1, \ldots, k$ are added to the tree. The second part of the test consists of m parallel threads, each adding an integer to the tree.

1 $\mathrm{Add}(1)$
2 $\ldots$
3 $\mathrm{Add}(k)$
4 $(\mathrm{Add}(k-\ell) \parallel \cdots \parallel \mathrm{Add}(k-\ell+\mathrm{m}))$

The second collection tests focuses on finding integers in the tree. Again, the first part of the test is sequential. In this part the integers $1, \ldots, k$ are added to the tree. The second part of the test consists of m parallel threads, each search for an integer in the tree.

1 $\mathrm{Add}(1)$
2 $\ldots$
3 $\mathrm{Add}(k)$
4 $(\mathrm{Contains}(k-\ell) \parallel \cdots \parallel \mathrm{Contains}(k-\ell+\mathrm{m}))$

The other tests remain to be discussed here.

5 Deadlocks, Uncaught Exceptions and Data Races

By default, JPF checks for deadlocks and uncaught exceptions. To also check for data races, we enable the listener PreciseRaceDetector, which is part of the package gov.nasa.jpf.listener. The tests described in Section 5 can be coded in Java in a straightforward way. JPF detected a race in the implementation of the red-black tree using fine-grain locking.

```
jpf +classpath=. +listener=gov.nasa.jpf.listener.PreciseRaceDetector Test1 1 0 2
JavaPathfinder v5.x - (C) RIACS/NASA Ames Research Center


====================================================== system under test
application: Test1.java
arguments:   1 0 2

====================================================== search started: 3/21/10 10:33 AM

====================================================== error #1
gov.nasa.jpf.listener.PreciseRaceDetector
race for field concurrent.RedBlackTree@299.root
  Thread-0 at concurrent.RedBlackTree.getRoot(RedBlackTree.java:29)
               "(concurrent/RedBlackTree.java:29)"  : getfield
  Thread-2 at concurrent.RedBlackTree.setRoot(RedBlackTree.java:39)
               "(concurrent/RedBlackTree.java:39)"  : putfield


====================================================== snapshot #1
thread index=1,name=Thread-0,status=RUNNING,this=Add@572,priority=5,
lockCount=0,suspendCount=0
  call stack:
        at concurrent.RedBlackTree.getRoot(RedBlackTree.java:29)
        at concurrent.RedBlackTree.add(RedBlackTree.java:88)
        at Add.run(Add.java:17)

thread index=3,name=Thread-2,status=RUNNING,this=Add@221,priority=5,
lockCount=0,suspendCount=0
  call stack:
        at concurrent.RedBlackTree.setRoot(RedBlackTree.java:39)
        at concurrent.RedBlackTree.add(RedBlackTree.java:255)
        at Add.run(Add.java:17)


====================================================== results
error #1: gov.nasa.jpf.listener.PreciseRaceDetector "race for field
concurrent.RedBlackTree@299.root   ..."

====================================================== statistics
elapsed time:       0:00:04
states:             new=2041, visited=2081, backtracked=4038, end=12
search:             maxDepth=125, constraints=0
choice generators:  thread=2040, data=0
heap:               gc=4844, new=855, free=1143
```

```
instructions:       120414
max memory:         33MB
loaded code:        classes=79, methods=1052

================================================== search finished: 3/21/10 10:33 AM
```

The race still remains to be discussed.

6 Post-Conditions

Describe the post-conditions to be checked. Discuss how they are checked. Discuss the results.

7 Size of the State Space

Give the size of the state space for the three different implementations for several tests.

8 Conclusion

Draw some conclusions.

References

[1] Franck van Breugel. Concurrent red-black trees, January 2010.

[2] Franck van Breugel. Implementing concurrent red-black trees in Java, March 2010.

[3] Gerard Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.

[4] Willem Visser, Klaus Havelund, Giullaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.