Verification of Java programs under fairness assumption

Vinh Cuong Tran, Hideki Hashimoto, Yoshinori Tanabe and Masami Hagiya
Graduate School of Information Science and Technology
The University of Tokyo

In this paper, we propose a method to verify Java programs under fairness assumption. Our fairness definition is described by Linear Time Temporal Logic (LTL) with propositions which express the facts that a thread can be selected to run and that it is actually executed. We have implemented the method as a library of Java PathFinder, an explicit-state model checker.

1 Introduction

Model checking is a technique for formally verifying finite-state concurrent systems. Model checking can be applied to both hardware and software, such as concurrent programs, to determine the validity of one or more properties of interest. Correctness properties which we want to model-check about concurrent programs fall into two broad classes : "safety" and "liveness" [4]. Safety properties, also known as "invariance" properties, intuitively assert that "nothing bad happens". Liveness properties are also referred to as "eventuality" properties or "progress" properties. Intuitively, a liveness property asserts that "something good will happen".

In this work, we focus on verifying liveness properties of Java programs. When liveness properties on concurrent systems are verified, it is often necessary to assume that the scheduling is *fair*. Therefore, we propose a definition of fairness in Java, and then implement a library that verifies properties based on the definition. Specifically, we have extended Java PathFinder (JPF) [1], a Java program model checker, for the purpose.

Our definition of fairness is based on Java virtual machine specification. In particular, it is defined based on the specification of lock and notification mechanisms of Java virtual machine (JVM). Furthermore, fairness assumption is expressed by using Linear Time Temporal Logic (LTL).

The rest of the paper is structured as follows. Section 2 gives formalization of fairness. Section 3 shows how verification of LTL formulae can be realized in JPF. Some verification examples and results are given in Section 4. Conclusion and future work are given in Section 5.

2 Fairness in Java programs

In this section, we give a definition of fairness for Java source code model checking.

There are three well-known definitions of fairness: unconditional fairness, weak fairness, and strong fairness [4]. The unconditional fairness, expressed by $\Box\Diamond(\textit{thread runs})$ in LTL, holds if every thread runs infinitely often. In Java programs, this kind of fairness is easily made false by having one thread loop infinitely after getting the common lock. Therefore, unconditional fairness is not appropriate for our purpose .

Weak fairness prohibits a case in which a thread is continuously enabled but is not executed. It is expressed as $\Box\Diamond((\textit{thread is enabled}) \rightarrow \textit{thread runs})$ in LTL. However, in Java, this is a too weak assumption. Consider a case in which two threads T1 and T2 run in infinite loops and they compete for a lock. A scheduling that allows T1 to acquire the lock every time is intuitively unfair. However, the weak fairness assumption does not exclude the scheduling, since T2 is not continuously enabled. In fact it is disabled when T1 acquires the lock.

The remaining fairness, strong fairness, is expressed by $\Box\Diamond\textit{thread is enabled} \rightarrow \Box\Diamond\textit{thread runs}$. It means if a thread is enabled infinitely often, it is executed infinitely often. It is the only fairness definition that can be employed for our purpose.

Now we investigate how the conditions *thread is enabled* and *thread runs* are to be defined. The definition of the latter should be clear, so we discuss the former.

Fairness is a condition on how threads are selected by the scheduler. In our framework, the scheduler selects a thread in the following three occasions.

1. Execution: since we work on the interleaving model, program execution is a loop in which the scheduler selects an enabled thread, which then executes an atomic operation.
2. Lock acquisition: if two or more threads compete for a lock, one of the threads is selected to acquire the lock.
3. Notification: if a *notify()* is issued by a thread

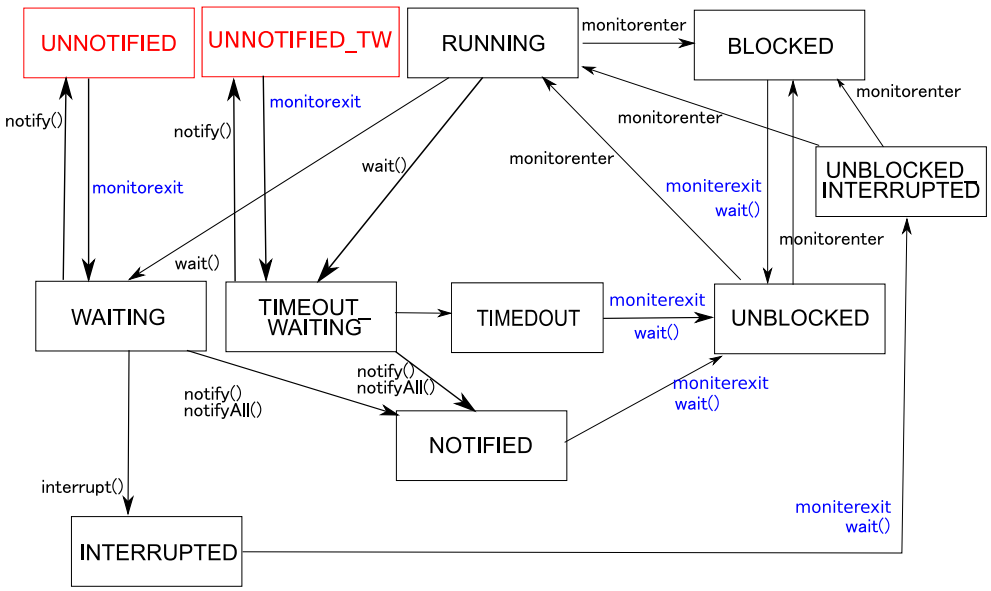

Fig. 1 Transition of thread statuses

Table. 1 Thread statuses

status	description
RUNNING	running
BLOCKED	blocked with requesting a lock
UNBLOCKED	the lock on which the thread was blocked is released
WAITING	issued *wait*()
TIMEOUT_WAITING	issued *wait*() with timeout
TIMEDOUT	*wait*() was timed out
NOTIFIED	notified by *notify*() or *notifyAll*()
INTERRUPTED	*wait*() was interrupted
UNBLOCKED_INTERRUPTED	*wait*() was interrupted and the lock was released
UNNOTIFIED	*notify*() was issued but the thread was not selected
UNNOTIFIED_TW	*notify*() was issued but the thread was not selected

that locks an object, and if two or more threads wait on the object, then one of the threads is selected to be notified.

The specification of JVM [8] defines the second and the third types of selections, and the first type of selection is due to our interleaving model. We investigate which thread is enabled for each of three types of selections by the scheduler listed above. As a basis of the investigation, we use the set of thread statuses in JPF. They are shown in Table 1 and Figure 1.

For the first type of selection, enabled threads have status RUNNING.

The second type is the lock acquisition. When a thread locks an object, the status of the locking thread is unchanged – it is RUNNING –, but the other threads that competed for the lock becomes BLOCKED. When the locking thread releases the lock, the thread status of the other threads is changed to UNBLOCKED, and they have a chance to acquire it. Therefore a thread can be regarded as “enabled” with respect to the second type of selection, if its status is either UNBLOCKED or RUNNING.

When the third type of selection takes place, only one waiting thread is notified. In JPF, the chosen thread changes its status to NOTIFIED, and continues to change to UNBLOCKED status when the monitor is released. The remaining threads that were not notified are kept in WAITING status. Although they were intuitively “enabled” when *notify*() is called, we cannot recognize this situation by observing their statuses defined so far. Therefore we introduce a new status called “UNNOTIFIED”. The status of a waiting thread is changed to UNNOTIFIED when *notify*() is called, and it goes back to WAITING immediately after that. Thus, we can detect the thread has been

enabled.

Similarly, we introduce a new status called "UNNOTIFIED_TW" for the status TIMEOUT_WAITING. The two new statuses are depicted as red rectangles in Figure 1.

The waiting threads can be interrupted, and change its status to INTERRUPTED. Then, when the lock is released its status becomes UNBLOCKED_INTERRUPTED, and it is ready to be scheduled again. Therefore, UNBLOCKED_INTERRUPTED should be taken similarly as UNBLOCKED.

In summary, we define the fairness condition now as

$$\bigwedge_{t \in Threads} (\Box\Diamond e_t \rightarrow \Box\Diamond r_t)$$

where

e_t = RUNNING $\vee$ UNBLOCKED $\vee$ UNBLOCKED_INTERRUPTED $\vee$ UNNOTIFIED $\vee$ UNNOTIFIED_TW for thread t, and

r_t = thread t actually runs.

3 LTL model checking under Java PathFinder

3.1 Java PathFinder

Java PathFinder is an explicit-state software model checker for Java bytecode. It takes as input a Java program, then searches for deadlocks and unhandled exceptions by default. But users can provide own property classes, or interfaces to verify other properties. The default search engine of JPF is only for verifying safety properties. We have to develop a new search engine for verifying liveness properties.

Our purpose is to verify liveness under fairness assumption, so we have two options. The first one is to develop a specialized algorithm for the purpose. The other option is to use LTL verifier. The first option will give more efficient implementation, but take more time to develop. The second option is more powerful since we can also verify a large class of LTL formulae, but is not efficient in general. We have chosen the second option, since we can use source code from our LTL verifier project.

3.2 Algorithm of LTL model checking in JPF

We employed the algorithm [7] for our search engine. It first builds finite-automaton on infinite words for the negation of the formula f. The resulting automaton is $A_{\neg f}$. Then, it computes the automaton for the program P. Finally, it checks if the language of the product of the two automata is empty. An acceptable run of the product is a counterexample of the property.

The algorithm [7] for checking satisfaction of LTL formula includes two depth-first searches (DFSs) as shown in Algorithms 1 and 2.

```
local q' ;
hash(q) ;
forall successors q' of q do
    if q' not in the hash table then
        outer-dfs (q') ;
    end
    if accept(q) then inner-dfs (q);
end
```

Algorithm 1: outer-dfs(q)

```
local q';
flag(q);
forall successors q' of q do
    if q' on outer-dfs stack then
        terminate (True);
    else if q' not flagged then
        inner-dfs (q');
    end
end
```

Algorithm 2: inner-dfs(q)

When the outer DFS is ready to backtrack from an accepting state after completing the search of its successors, the inner DFS will begin to find a loop through this state. In case that the inner DFS fails to find a loop, the outer DFS resumes from the point where it was interrupted. The `terminate`($True$) is called when the loop has been found.

JPF provides *forward()* and *backtrack()* methods of the parent class Search for implementing customized search engine. When *forward()* is called it will check if there is any successor that has not been explored, and return true if it is the case. The information about which successors have been visited is contained in an object called ChoiceGenerator. Each program state is visited only once in the original search engine of JPF. That is not the case for the algorithm we use. In particular, when the inner search starts, we need to explore the successors of a state which the outer DFS has just finished exploring. In order to do this, we manage to reset the ChoiceGenerator object so that it repeats exploration.

3.3 Implementation

In order to implement the algorithm described in Section 3.2, we have used LTL2BA4J [3], a version of ltl2ba [2] for Java. ltl2ba is a tool written

in ANSI C, based on paper [5], which allows conversion of formulae in the Linear Time Temporal Logic to Büchi automata.

In theory, each state of Kripke structure has an associated set of propositions, which are true at that state. In JPF, the states are of programs only generated when nondeterminism encountered, so that it is difficult to determine when a new state is created.

We have to determine the truth values of propositions when a new state is created. If the value of a proposition can be determined from the information of the current state of system, there should be no problem. But we need to enable propositions whose values depend on history. This is because the states are only sparsely created. For example, if we want to express $\Box$(*method1() is called*), we need to have a proposition that becomes true when *method1()* has been called since the previous state was created. Therefore, we have to keep track of the execution of the program from the previous state to the current state. Also the value of such a proposition has to be reset whenever the execution from the current state begins, otherwise, the value of the proposition of the next state will be the same as that of the previous state.

We created an interface called Proposition, and each proposition will implement this interface to specify what the proposition is about. Some of the methods of the interface are *getValue()* method, for search engine to get the value of proposition, and *reset()* method for resetting the proposition as described above. In order to ease the implementation of propositions, we also provided PropositionListenerAdapter that implemented all the methods of interface Proposition. The name of the proposition, e.g. p, q, is mapped with the name of the class in configuration file, such as in Figure 2. The formula to be verified is also specified in the configuration file.

A sample of implementation of proposition is given in Figure 3 and 4. The source in Figures 3 and 4 illustrate how to implement proposition e (*thread is enabled*), and r (*thread run*), in our definition of fairness. These two classes are provided as built-in classes. The user only have to provide implementations for the propositions in liveness properties, which can be implemented the same way as those of Figure 3 and 4. Although, we have defined the fairness condition for all threads of the program, in Figure 3, we only specified the *enabled* condition for thread 1.

Generally speaking, LTL model checking assumes that the transition system under verification does not has end states. However, naturally, some Java programs terminate, meaning

```
jpf.listener=jpf.ltl.GeneralProperty
search.class=jpf.ltl.LTLSearch
#ltl.formula = []( p -> <>q )
ltl.formula = ([]<>e -> []<>r) -> []( p -> <>q )
ltl.proposition.e = sample.ReadyToRunProp
ltl.proposition.r = sample.ActuallyRunProp
ltl.proposition.p = sample.RequestLockProp
ltl.proposition.q = sample.AcquireLockProp
```

Fig. 2 Configuration file

```
import gov.nasa.jpf.jvm.ThreadInfo;
import gov.nasa.jpf.jvm.ThreadList;
import gov.nasa.jpf.search.Search;
import jpf.ltl.PropositionListenerAdapter;

public class ReadyToRunProp extends PropositionListenerAdapter{
 private boolean value;

 public ReadyToRunProp() {
  value = false;
 }

 public boolean getValue(JVM vm, Search search) {
  return value;
 }

 public void reset(JVM vm, Search search) {
  value = false;
 }

 public void restore(boolean previousValue, JVM vm, Search search) {
  value = previousValue;
 }

 public void instructionExecuted(JVM vm) {
  if (value) { return; }
  ThreadList tl = vm.getThreadList();
  if (tl.length() <= 1) { return; }
  ThreadInfo ti = tl.get(1);
  switch(ti.getStatus()) {
  case ThreadInfo.RUNNING:
  case ThreadInfo.UNBLOCKED:
  case ThreadInfo.UNBLOCKED_INTERRUPTED:
   value = true;
   break;
  }
 }
}
```

Fig. 3 Sample implementation of e proposition

their state space have end states. We avoid this problem by adding a self loop to each end state.

4 Model checking results

In this section, we provide two sample programs to test the two selections of fairness. In both samples, we verify a liveness property which asserts that “whenever a thread requests a lock, it will eventually acquires the lock”. The first sample in Figure 5 (called sample 5) is for testing strong fairness of JVM’s lock mechanism, where a synchronized statement is used for the lock. The second sample in Figure 6 (called sample 6) is used to verify the strong fairness assumption of notification mechanism, where *wait()* and *notify()* are used.

In the two samples, we use *Verify.getInt(int min,*

```
package sample;
import gov.nasa.jpf.jvm.JVM;
import gov.nasa.jpf.search.Search;
import jpf.ltl.Proposition.ListenerAdapter;

public class ActuallyRunProp extends PropositionListenerAdapter{
 private boolean value;

 public ActuallyRunProp() {
  value = false;
 }

 public boolean getValue(JVM vm, Search search) {
  return value;
 }

 public void reset(JVM vm, Search search) {
  value = false;
 }

 public void restore(boolean previousValue, JVM vm, Search search) {
  value = previousValue;
 }

 public void instructionExecuted(JVM jvm) {
  if (jvm.getCurrentThread().getIndex() == 1) { value = true; }
 }
}
```

Fig. 4 Sample implementation of r proposition

int max). Verify.getInt(int min, int max) is a method of the Verify class that is included in JPF. It returns an integer nondeterministically between (and including) min and max. Therefore, when search engine encounter this statement, the new program state will be created for each number.

Sample 5 has two threads trying to call method *acquireLock()* after calling *requestLock(). acquireLock()* and *requestLock()* are just two dummy methods created for testing liveness properties. *Verify.getInt(0,1)* will return number 0 or 1, and just when 0 is returned the thread can execute the body of *run()*. There is a case that when *c* in both threads equals 0, and both threads compete for the implicit lock. If fairness is not assumed, the error will be reported.

Also in sample 5, if we uncomment the while loop in *aquireLock*, we have another test sample. This test sample creates a situation in which one thread will loop forever in the uncommented loop, so that it has an error even if fairness was assumed.

Sample 6 is different from sample 5 only in notification mechanism is used. In sample 6 a thread will get in a wait set and wait until it is notified. If no fairness is assumed, there is possible that one thread will wait forever.

We performed a test with out current implementation. The results are shown in Table 2. All results are as expected.

5 Conclusion and future work

5.1 Conclusion

The main contribution of this work is twofold. We have proposed a definition of fairness in Java,

```
package sample;
import gov.nasa.jpf.jvm.Verify;

public class LockTest extends Thread {
   static Object lock = new Object();
   public void run() {
      while (true) {
         int c = Verify.getInt(0,1);
         if (c == 0) {
            requestLock();
            synchronized(lock) {
               acquireLock();
            }
         }
      }
   }
   public static void main(String[] args) {
      LockTest th1 = new LockTest();
      LockTest th2 = new LockTest();
      th1.start();
      th2.start();
   }
   static public void requestLock() {}
   static public void acquireLock() {
      /*
      while (true) {
         int d = Verify.getInt(0,1);
         if (d == 0) {
            break;
         }
      }
      */
   }
}
```

Fig. 5 sample program for demonstration the fairness of synchronized structure

and implemented a library with which the user can verify properties under fairness assumption based on the definition.

Our definition of fairness covers selections of thread in competing locks and in notification.

5.2 Future work

Although we confirmed that our definition of fairness worked fine with sample programs, we should further apply it to real programs. For that purpose, more efficient implementation will be needed.

Verification under strong fairness assumption is not so efficient that it is desirable to avoid the assumption. One of the idea to achieve that is to use an alternative library for lock, unlock, and notify, which implements them in first-in first-out manner. In such an implementation, weak fairness and strong fairness conditions coincide, therefore it should be able to apply known efficient algorithms for weak fairness.

References

[1] Java pathfinder. `http://javapathfinder.sourceforge.net/`.

```
package sample;
import gov.nasa.jpf.jvm.Verify;

public class NotifyTest extends Thread {
 static Object lock = new Object();
 private int type;

 public NotifyTest(int type) {
  this.type = type;
 }

 public void run() {
  switch(type) {
  case 0:   run0();   break;
  case 1:   run1();   break;
  case 2:   run2();   break;
  default:  assert false : "internal error";
  }
 }

 public void run0() {
  while (true) {
   int c = Verify.getInt(0,1);
   if (c == 0) {
    requestLock();
    synchronized(lock) {
     try {
        lock.wait();
     } catch (InterruptedException e) {
      e.printStackTrace();
     }
     acquireLock();
    }
   }
  }
 }

 public void run1() {
  while (true) {
   synchronized(lock) {
    try {
     lock.wait();
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
   }
  }
 }

 public void run2() {
   while (true) {
    synchronized(lock) {
     lock.notify();
    }
   }
  }

 public static void main(String[] args) {
   NotifyTest th1 = new NotifyTest(0);
   NotifyTest th2 = new NotifyTest(1);
   NotifyTest th3 = new NotifyTest(2);
   th1.start();
   th2.start();
   th3.start();
  }

  static public void requestLock() {}

  static public void acquireLock() {
   while (true) {
    int d = Verify.getInt(0,1);
    if (d == 0) {
     break;
    }
   }
  };
 }
```

Fig. 6 Sample program for demonstration the fairness of notification

	$\Box(requestLock \rightarrow \Diamond acquireLock)$	$(\Box\Diamond e \rightarrow \Box\Diamond r) \rightarrow$ $\Box(requestLock \rightarrow \Diamond acquireLock)$
sample 5	error	satisfied
sample 5 with *acquireLock()* contains infinite loop	error	error
sample6	error	satisfied

Table. 2 Model checking results

[2] ltl2ba. `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php`.

[3] Ltl2ba4j. `http://www-i2.informatik.rwth-aachen.de/Forschung/RV/ltl2ba4j/index.html`.

[4] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

[5] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer.

[6] Dimitra Giannakopoulou, Jeff Magee, and Jeff Kramer. Fairness and priority in progress property analysis, 1999.

[7] Gerard Holzmann, Doron Peled, and Mihalis Yannakakis. On nested depth first search. In *In The Spin Verification System*, pages 23–32. American Mathematical Society, 1996.

[8] Tim Lindholm and Frank Yellin. The java virtual machine specification. Website, 1999. `http://java.sun.com/docs/books/jvms/`.