Using Runtime Analysis to Guide Model Checking of Java Programs

Klaus Havelund

QSS/Recom
NASA Ames Research Center
Moffett Field, CA, USA
havelund@ptolemy.arc.nasa.gov,
http://ase.arc.nasa.gov/havelund

Abstract. This paper describes how two runtime analysis algorithms, an existing data race detection algorithm and a new deadlock detection algorithm, have been implemented to analyze Java programs. Runtime analysis is based on the idea of executing the program once, and observing the generated run to extract various kinds of information. This information can then be used to predict whether other different runs may violate some properties of interest, in addition of course to demonstrate whether the generated run itself violates such properties. These runtime analyses can be performed stand-alone to generate a set of warnings. It is furthermore demonstrated how these warnings can be used to guide a model checker, thereby reducing the search space. The described techniques have been implemented in the home grown Java model checker called Java PathFinder.
Keywords Concurrent programs, runtime analysis, race conditions, deadlocks, program verification, guided model checking, Java.

1 Introduction

Model checking of programs has received an increased attention from the formal methods community within the last couple of years. Several systems have emerged that can model check source code, such as Java, C and C++ directly (typically subsets of these languages) [17, 9, 4, 20, 28, 25]. The majority of these systems can be classified as *translators*, which translate from the programming language source code to the modeling language of the model checker. The Java PathFinder 1 (JPF1) [17], developed at NASA Ames Research Center, was such an early attempt to bridge the gap between Java [12] and the PROMELA language of SPIN [21]. A second generation of Java PathFinder (JPF2) [28] has recently been developed at NASA Ames, which diverges from the translation approach, and model checks bytecode directly. This system contains a home grown Java Virtual Machine (JVM) specifically designed to support memory efficient storage of states for the purpose of model checking. This system resembles the Rivet machine described in [3] in the sense that Rivet also provides its own new JVM.

The major obstacle for model checking to succeed is of course the management of large state spaces. For this purpose abstraction techniques have been studied heavily in the past 5 years [18, 2, 13, 8, 1]. More recently, special focus has been put on abstraction environments for Java and C [5, 6, 29, 20, 14, 25]. Alternatives to model checking have also been tried, such as VeriSoft [11], which performs stateless model checking of C++ programs, and ESC [10], which uses a combination of static analysis and theorem proving to analyze Modula3 programs. Of course static program analysis techniques [7] is an entire separate discipline, although it yet remains to be seen how well they can handle concurrency. An alternative to the above mentioned techniques is *runtime analysis*, which is based on the idea of concluding properties of a program from a single run of the program. Hence, executing the program once, and observing the run to extract information, which is then be used to predict whether other different runs may violate some properties of interest (in addition of course to demonstrate whether the generated run violates such properties). The most known example of a runtime analysis algorithm is the data race detection algorithm Eraser [26], developed by S. Savage, M. Burrows, G. Nelson, and P. Sobalvarro. A data race is the simultaneous access to an unprotected variable by several threads. An important characteristic of this algorithm is that a run itself does not have to contain a data race in order for data races in other runs to be detected. This kind of algorithm will not guarantee that errors are found since it works on an arbitrary run. It may also may yield false positives. What is attractive, however, that the algorithm scales very well since only one run needs to be examined. Also, in practice Eraser often seems to catch the problems it is designed to catch independently of the run chosen. That is, the arbitrariness of the chosen run does not seem to imply a similar arbitrariness in the analysis results.

The work presented in this paper describes an extension to JPF2 to perform runtime analysis on multi-threaded Java programs in simulation mode, either stand-alone, or as a pre-run to a subsequent model checking, which is guided by the warnings generated during the runtime analysis. We implement the generic Eraser algorithm to work for Java, and furthermore develop and implement a new runtime analysis algorithm, called GoodLock, that can detect deadlocks. We furthermore implement a third runtime dependency analysis used to do dynamic slicing of the program before the model checker is activated on the set of runtime analysis warnings. Section 2 describes the Eraser algorithm from [26], and how it is implemented in JPF2 to work on Java programs. Section 3 describes the new GoodLock algorithm and it implementation. Section 4 describes how these analyses, in addition to being run stand alone, can be performed in a pre-run to yield warnings, that are then used to guide a model checker. This section includes a presentation of a runtime dependency analysis algorithm to reduce the program to be model checked. Finally, Section 6 contains conclusions and a description of future work.

2 Data Race Detection

This section describes the Eraser algorithm as presented in [26], and how it has been implemented in JPF2 to work on Java programs. A *data race* occurs when two concurrent threads access a shared variable and when at least one access is a *write*, and the threads use no explicit mechanism to prevent the accesses from being simultaneous. The Eraser algorithm detects data races in a program by studying a single run of the program, and from this trying to conclude whether any runs with data races are possible. We have implemented the generic Eraser algorithm described in [26] to work for Java's synchronization constructs. Section 2.1 illustrates with an example how JPF2 is run in Eraser mode. Section 2.2 describes the generic Eraser algorithm, while Section 2.3 describes our implementation of it for Java.

2.1 Example

The Java program in Figure 1 illustrates a potential data race problem.

```
 1. class Value{
 2.   private int x = 1;
 3.
 4.   public synchronized void add(Value v){x = x + v.get();}
 5.
 6.   public int get(){return x;}
 7. }
 8.
 9. class Task extends Thread{
10.   Value v1; Value v2;
11.
12.   public Task(Value v1,Value v2){
13.     this.v1 = v1; this.v2 = v2;
14.     this.start();
15.   }
16.
17.   public void run(){v1.add(v2);}
18. }
19.
20. class Main{
21.   public static void main(String[] args){
22.     Value v1 = new Value(); Value v2 = new Value();
23.     new Task(v1,v2); new Task(v2,v1);
24.   }
25. }
```

Fig. 1. Java program with a data race condition.

Three classes are defined: The `Value` class contains an integer variable that is accessed through two methods. The `add` method takes another `Value` object as parameter and adds the two, following a typical object oriented programming style. The method is synchronized, which means that when called by a thread, no other thread can call synchronized methods in the same object. The `Task` class inherits from the system defined `Thread` class, and contains a constructor (lines 12-15) that is called when objects are created, and a `run` method that is called when these objects are started with the start method. Finally, the main method in the Main class starts the program. When running JPF2 in simulation mode with the Eraser option switched on, a data race condition is found, and reported as illustrated in Figure 2.

```
******************************
Race condition!
------------------------------
Variable x in class Value
is accessed unprotected.
******************************
From Task thread:
--------------------------
read access
Value.get line 6
Value.add line 4
Task.run line 17

From Task thread:
--------------------------
write access
Value.add line 4
Task.run line 17
=============================
```

Fig. 2. Output generated by JPF2 in Eraser simulation mode.

The report tells that the variable x in class Value is accessed unprotected, and that this happens from the two Task threads, from lines 4 and 6, respectively, also showing the call chains from the top-level run method. The problem detected is that one Task thread can call the add method on an object, say v1, which in turn calls the unsynchronized get method in the other object v2. The other thread can simultaneously make the dual operation, hence, call the add method on v2. Note that the fact that the add method is synchronized does not prevent its simultaneous application on two different Value objects by two different threads.

2.2 Algorithm

The basic algorithm works as follows. For each variable x, a set of locks $\mathbf{set}(x)$ is maintained. At a given point in the execution, a lock l is in $\mathbf{set}(x)$ if each thread that has accessed x held l at the time of access. As an example, if one thread has the lock l_1 when accessing a variable x, and another thread has lock l_2, then $\mathbf{set}(x)$ will be empty after those two accesses, since there are no locks that both threads have when they access the variable. If the set in this way becomes empty, it means that there does not exist a lock that protects it, and a warning can be issued, signaling a potential for a data race.

The set of locks protecting a variable can be calculated as follows. For each thread t is maintained the set, $\mathbf{set}(t)$, of locks that the thread holds at any time. When a thread for example calls a synchronized method on an object, then the thread will lock this object, and the set will be updated. Likewise, when the thread leaves the method, the object will be removed from the lock set, unless the thread has locked the object in some other way. When the thread t accesses a variable x (except for the first time), the following calculation is then performed:

$\text{set}(x) := \text{set}(x) \cap \text{set}(t);$
$\textbf{if } \text{set}(x) = \{\} \textbf{ then } \text{issue warning}$

The lock set associated to the variable is *refined* by taking the intersection with the set of locks held by the accessing thread. The initial set, $\mathbf{set}(x)$, of locks of the variable x is in [26] described to be the set of all locks in the program. In a Java program objects (and thereby locks) are generated dynamically, hence the set of all locks cannot be pre-calculated. Instead, upon the first access of the variable, $\mathbf{set}(x)$ is assigned the set of locks held by the accessing thread, hence $\mathbf{set}(t)$.

The simple algorithm described above yields too many warnings as explained in [26]. First of all, shared variables are often *initialized* without the initializing thread holding any locks. In Java for example, a thread can create an object by the statement `new C()`, whereby the `C()` constructor will initialize the variables of the object, probably without any locks. The above algorithm will yield a warning in this case, although this situation is safe. Another situation where the above algorithm yields unnecessary warnings is if a thread creates an object, where after several other threads read the object's variables (but no-one is writing after the initialization).

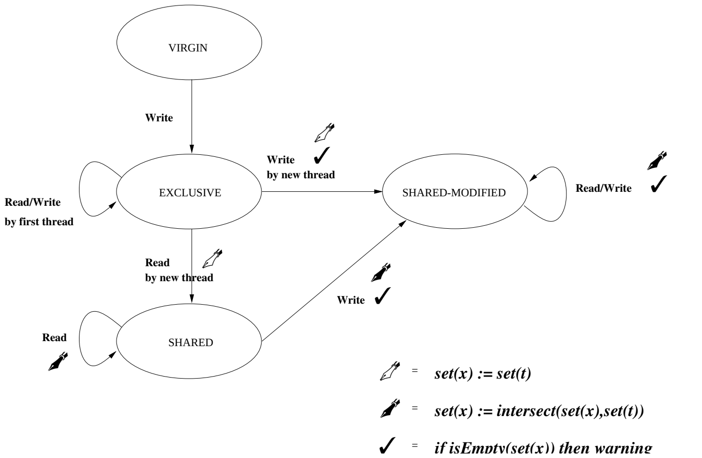

Fig. 3. The Eraser algorithm associates a state machine with each variable x. The state machine describes the Eraser analysis performed upon access by any thread t. The pen heads signify that lock set refinement is turned on. The "ok" sign signifies that warnings are issued if the lock set becomes empty.

To avoid warnings in these two cases, [26] suggests to extend the algorithm by associating a state machine to each variable in addition to the lock set. Figure 3 illustrates this state machine. The variable starts in the VIRGIN state. Upon the first write access to the variable, the EXCLUSIVE state is entered. The lock set of the variable is not refined at this point. This allows for initialization without locks. Upon a read access by another thread, the SHARED state is entered, now with the lock refinement switched on, but without yielding warnings in case the lock set goes empty. This allows for multiple readers (and not writers) after the initialization phase. Finally, if a new thread writes to the variable, the SHARED-MODIFIED state is entered, and now lock refinements are followed by warnings if the lock set becomes empty.

2.3 Implementation

The Eraser algorithm has been implemented by modifying the home grown Java Virtual machine to perform this analysis when the `eraser` option is switched on. Two new Java classes are defined: `LockSet`, implementing the notion of a set of locks, and `LockMachine`, implementing the state machine and lock set, that is associated with each variable.

Lock Sets Associated with Threads Each thread is associated with a `LockSet` object, which is updated whenever a lock on an object is taken or released. The interface of this class is:

```
interface iLockSet{
  void    addLock(int objref);
  void    deleteLock(int objref);
  void    intersect(iLockSet locks);
  boolean contains(int objref);
  boolean isEmpty();
}
```

This happens for example when a `synchronized` statement such as:

```
synchronized(lock){
  ...
}
```

is executed. Here `lock` will refer to an object, the object reference of which will then be added to the lock set of the thread that executes this statement. Upon exit from the statement, the lock is removed from the thread's lock set, if the lock has not been taken by an enclosing synchronized statement. This can occur for example in a statement like[1]:

```
synchronized(lock){
  synchronized(lock){
    ...
  };
  (*)
}
```

[1] This statement illustrates a principle and does not represent a programming practice.

In this case, leaving the inner `synchronized` statement should not cause the lock to be removed from the thread's lock set since the outer statement still causes the lock to be held at point (*). The JPF2 JVM already maintains a counter that tracks the nesting, and this counter is then used to update the lock sets correctly. Note that conceptually a synchronized method such as:

```
public synchronized void doSomething(){
  ...
}
```

can be regarded as short for:

```
public void doSomething(){
  synchronized(this){
    ...
  }
}
```

State Machines Associated with Variables The `LockMachine` class has the following interface:

```
interface iLockMachine{
  void checkRead(ThreadInfo thread);
  void checkWrite(ThreadInfo thread);
}
```

An object of the corresponding class is associated to each variable, and its methods are called whenever a variable field is read from or written to. Variables include instance variables as well as static variables of a class, but not variables local to methods since these cannot be shared between threads.

Instrumenting the Bytecodes A Java program is translated into bytecodes by the compiler. The bytecodes manipulate a stack of method frames, each with an operand stack. Objects are stored in a heap. The `add` method of the `Value` class in Figure 1, for example, is by the Java compiler translated into the following bytecodes:

```
Method void add(Value)
   0 aload_0
   1 aload_0
   2 getfield #7 <Field int x>
   5 aload_1
   6 invokevirtual #6 <Method int get()>
   9 iadd
  10 putfield #7 <Field int x>
  13 return
```

The reference (*this*) of the object on which the `add` method is called, is loaded twice on the stack (lines 0 and 1), wherafter the `x` field of *this* object is extracted by the `getfield` bytecode, and put on the stack, replacing the topmost *this* reference. The object reference of the argument `v` is then loaded on the stack (line 5), and the `get` method is called by the `invokevirtual` bytecode, the result being stored on the stack. Finally the results are added and restored in the `x` field of *this* object.

The JPF2 JVM accesses the bytecodes via the JavaClass package [23], which for each bytecode delivers a Java object of a class specific for that bytecode (recall that JPF2 itself is written in Java). The JPF2 JVM extends this class with an `execute` method, which is called by the verification engine, and which represents the semantics of the bytecode. The runtime analysis is obtained by further annotating the `execute` method. For example, a `getfield` bytecode is delivered to the JPF2 JVM as an object of the following class, containing an `execute` method, which makes a conditional call (if the Eraser option is set) of the `checkRead` method of the lock machine of the variable being read.

```
public class GETFIELD extends AbstractInstruction {
  public InstructionHandle execute(SystemState s) {
    ...

    if (Eraser.on){
      da.getLockStatus(objref,fieldName).checkRead(th);
    }

    ...
  }
}
```

A similar annotation is made for the `PUTFIELD` bytecode. Similar annotations are also made for static variable accesses such as the bytecodes `GETSTATIC` and `PUTSTATIC`, and all array accessing bytecodes such as for example `IALOAD` and `IASTORE`. The bytecodes `MONITORENTER` and `MONITOREXIT`, generated from explicit `synchronized` statements, are annotated with updates of the lock sets of the accessing threads to record which locks are owned by the threads at any time; just as are the bytecodes `INVOKEVIRTUAL` and `INVOKESTATIC` for calling synchronized methods. The `INVOKEVIRTUAL` bytecode is also annotated to deal with the built-in `wait` method, which causes the calling thread to release the lock on the object the method is called on. Annotations are furthermore made to bytecodes like `RETURN` for returning from synchronized methods, and `ATRHOW` that may cause exceptions to be thrown within synchronized contexts.

3 Deadlock Detection

In this section we present a new runtime analysis algorithm, called GoodLock, for detecting deadlocks. A classical deadlock situation can occur where two threads share two locks, and they take the locks in different order. This is illustrated in Figure 4, where thread 1 takes the lock Ll first, while thread 2 takes the lock L2 first, wherafter, each of the two threads is now prevented from getting the remaining lock because the other thread has it.

3.1 Example

To demonstrate this situation in Java, suppose we want to correct the program in Figure 1, eliminating the data race condition problem by making the `get` method synchronized, as shown in Figure 5, line 6 (we just add the `synchronized` keyword to the method signature).

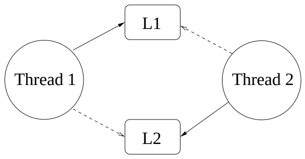

Fig. 4. Classical deadlock where task 1 takes lock Ll first and task 2 takes lock L2 first.

```
1. class Value{
2.   private int x = 1;
3.
4.   public synchronized void add(Value v){x = x + v.get();}
5.
6.   public synchronized int get(){return x;}
7. }
```

Fig. 5. Avoiding the data race condition by making the `get` method synchronized.

Now the `x` variable can no longer be accessed simultaneously from two threads, and the Eraser module will no longer give a warning. When running JPF2 in simulation mode with the GoodLock option switched on, however, a lock order problem not present before is now found, and reported as illustrated in Figure 6.

```
******************************
Lock order conflict!
------------------------------
Locks on Value#1 and Value#0
are taken in opposite order.
******************************
Lock on Value#1 is taken last
by Task thread:
------------------
Value.add line 4
Task.run line 17

Lock on Value#0 is taken last
by Task thread:
------------------
Value.add line 4
Task.run line 17
==============================
```

Fig. 6. Output generated by JPF2 in GoodLock simulation mode.

The report explains that the two object instances of the `Value` class, identified by the internal object numbers `#0` and `#1`, are taken in a different order by the two `Task` threads, and it indicates the line numbers where the threads may deadlock, hence where the access to the second lock may fail. That is, line 4 contains the call of the `get` method from the `add` method. The problem arises due to the fact that the `get` method has become synchronized. One task may now call the `add` operation on a `Value` object, say `v1`, which in turn calls the `get` method on the other object `v2`; hence locking `v1` and then `v2` in that order.

Since the other task will do the reverse, we have a situation as illustrated in Figure 4.

An algorithm that detects such lock cycles must in addition take into account that a third lock may protect against a deadlock like the one above, if this lock is taken as the first thing by both threads, before any of the other two locks are taken. In this situation no warnings should be emitted. Such a protecting third lock is called a *gate lock*. The algorithm below does *not* warn about a lock order problem in case a gate lock prevents the deadlock from ever happening.

3.2 Algorithm

The algorithm for detecting this situation is based on the idea of recording the locking pattern for each thread during runtime as a lock tree, and then when the program terminates to compare the trees for each pair of threads as explained below. If the program does not terminate by itself, the user can terminate the execution by a single stroke on the keyboard, when he or she believes enough information has been recorded, which can be inferred by information being printed out. The lock tree that is recorded for a thread represents the nested pattern in which locks are taken by the thread. As an artificial example, consider the code fragments of two threads in Figure 7. Each thread executes an infinite loop, where in each iteration four locks, L1, L2, L3 and L4, are taken and released in a certain pattern. For example, the first thread takes L1; then L3; then L2; then it releases L2; then takes L4; then releases L4; then releases L3; then releases L1; then takes L4; etc.

```
Thread 1: while(true){
            synchronized(L1){
              synchronized(L3){
                synchronized(L2){};
                synchronized(L4){}
              }
            };
            synchronized(L4){
              synchronized(L2){
                synchronized(L3){}
              }
            }
          }
```

```
Thread 2: while(true){
            synchronized(L1){
              synchronizd(L2){
                synchronized(L3){}
              }
            };

            synchronized(L4){
              synchronized(L3){
                synchronized(L2){}
              }
            }
          }
```

Fig. 7. Synchronization behavior of two threads.

This pattern can be observed, and recorded in a finite tree of locks for each thread, as shown in Figure 8, by just running the program for a large enough period to allow both loops to be iterated at least once. As can be seen from the tree, a deadlock is potential because thread 1 in its left branch locks L3 (node identified with 2) and then L4 (4), while thread 2 in its right branch takes these locks in the opposite order (11, 12). There are furthermore two additional ordering problems between L2 and L3, one in the two left branches (2, 3 and 9, 10), and one in the two right branches (6, 7 and 12, 13). However, neither of these pose a deadlock problem since they are protected by the *gate locks* L1 (1, 8) respectively L4 (5, 11). Hence, one warning should be issued, corresponding to the fact that this program would deadlock if thread 1 takes lock L3 and thread 2 takes lock L4.

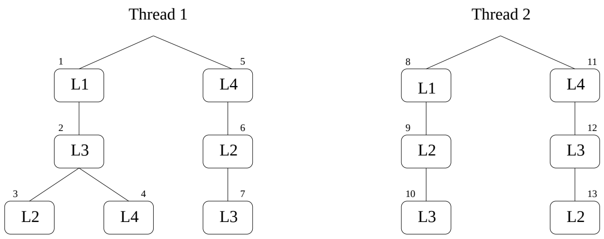

Fig. 8. Lock trees corresponding to threads in Figure 7.

The tree for a thread is built as follows. Each time an object o is locked, either by calling a synchronized method m on it, as in $o.m(\ldots)$, or by executing a statement of the form: `synchronized(`o`)`$\{\ldots\}$, the 'lock' operation in Figure 9 is called. Likewise, when a lock is released by the return from a synchronized method, or control leaves a `synchronized` statement, the 'unlock' operation is called. The tree has at any time a *current node*, where the path from the root (identifying the thread) to that node represents the *lock nesting* at this point in the execution: the locks taken, and the order in which they were taken. The lock operation creates a new child of the the current node if the new lock has not previously been taken with that lock nesting. The unlock operation just backs up the tree if the lock really is released, and not owned by the thread in some other way. For the program in Figure 7, the trees will stabilize after one iteration of each loop, and will not get updated further. A print statement can inform the user whenever a new lock pattern is recognized and thereby a tree is updated, thereby making it easier for the user to decide when to terminate the program in case it is infinitely looping (if nothing is printed out after a while it is unlikely that new updates to the tree will occur).

```
lock(Thread thread,Lock lock){
    if thread does not already own lock{
        if lock is a son of current{
            current = that son
        }else{
            add lock as a new son of current;
            current = new son;
            print("new pattern identified");}}}

unlock(Thread thread,Lock lock){
    if thread does not own lock in another way{
        current = parent of current node;}}
```

Fig. 9. Operations 'lock' and 'unlock' used for creating a lock tree.

When the program terminates, the analysis of the lock trees is initiated by a call of the 'analyze' operation in Figure 10. This operation compares the trees for each pair of threads[2]. For each pair (t_1, t_2) of trees, such as those in Figure 8, the operation 'analyzeThis' is called recursively on all the nodes n_1 in t_1; and for every node n_2 in t_2 with the same lock as n_1, it is checked that no lock below n_1 in t_1 is above n_2 in t_2. In order to avoid issuing warnings when a *gate lock* prevents a deadlock, nodes in t_2 are marked after being examined, and nodes below marked nodes are not considered until the marks are removed when the analyzeThis operation backtracks from the corresponding node in t_1. This will prevent warnings from being issued about locks L2 and L3 in Figure 8, since the nodes 8 and 11 of thread 2 will get marked, when the trees below nodes 1 respectively 3 in thread 1 get examined. This reflects that nodes L1 and L4 are such gate locks preventing deadlocks due to lock order conflicts lower down the trees.

```
analyze(){
    for each pair (t1,t2) of thread trees{
        for each immediate child node n1 of t1's topnode{
            analyzeThis(n1,t2);}}}

analyzeThis(LockNode n,LockTree t){
    Set N = {nt ∈ t | nt.lock == n.lock ∧ nt is not below a mark};
    for each nt in N{
        check(n,nt);
    };
    mark nodes in N;
    for each child n^child of n{
        analyzeThis(n^child,t);
    };
    unmark nodes in N;}

check(n1,n2){
    for each child node n1^child of n1{
        if n1^child.lock is above n2{
            conflict()
        }else{
            check(n1^child,n2)}}}
```

Fig. 10. Operations 'analyze', 'analyzeThis', and 'check' used for analyzing lock trees.

The program in Figure 1 with the change indicated in Figure 5 has a potential for deadlock, which is detected by the GoodLock algorithm since each of the lock trees describes two locks on Value objects taken one after the other, but in different order in the two trees. Note, however, that the detection of a deadlock potential is not a proof of the existence of a deadlock. The program may prevent the deadlock in some other way. It is just a warning, which may focus our attention towards a potential problem. Note also, that the algorithm as described only detects deadlock potentials between pairs of threads. That is, although the analyzed program can have a very large number of threads, which is the

[2] The operation is symmetric such that only one ordering of a pair needs to be examined.

major strength of the algorithm, deadlocks will only be found if they involve two threads. A generalization is needed to identify deadlocks between more than two threads. The generalization must identify a subset of threads (trees) which together create a conflict. Consider for example three threads, each taking 2 out of 3 locks L1, L2 and L3 as follows: <L1,L2>, <L2,L3> and <L3,L1>. One can easily detect this deadlock by observing that as their first steps they together take all the locks, which prevent them from taking their second step each.

3.3 Implementation

The major new Java class defined is `LockTree`, which describes the lock tree objects that are associated with threads, and that are updated during the runtime analysis, and finally analyzed after program termination. Its interface is:

```
interface iLockTree{
  void lock(Lock lock);
  void unlock();
  void analyze(iLockTree otherTree);
}
```

The following bytecodes will activate calls of the lock and unlock operations in these tree objects for the relevant threads: MONITORENTER and MONITOREXIT for entering and exiting monitors, INVOKEVIRTUAL and INVOKESTATIC for calling synchronized methods or the built-in `wait` method of the Java threading library, bytecodes like RETURN for returning from synchronized methods, and ATRHOW that may cause exceptions to be thrown within synchronized contexts. Methods are in addition provided for printing out the lock trees, a quite useful feature for understanding the lock pattern of the threads in a program.

4 Integrating Runtime Analysis with Model Checking

The runtime analyses as described in the previous two sections can provide useful information to a programmer as stand alone tools. In this section we will describe how runtime analysis furthermore can be used to guide a model checker. The basic idea is to first run the program in simulation mode, with all the runtime analysis options turned on, thereby obtaining a set of warnings about data races and lock order conflicts. The threads causing the warnings, called the *race window*, is then fed into the model checker, which will then focus it attention on the threads that were involved in the warnings. For this to work, the race window often must be extended to include threads that create or otherwise influence the threads in the original window. A runtime dependency analysis is used as a basis for this extension of the race window.

4.1 Example

Consider the program in Figure 1, troubled by a deadlock potential caused by the change indicated in Figure 5. If, instead of applying the runtime analysis, we apply the JPF2 model checker to this program, the deadlock is immediately found and reported via an error trail leading from the initial state to the deadlocked state. Suppose, however, that this program is a subprogram of a larger program that spawns other threads not influencing the behavior of the two tasks involved in the deadlock. In this case the model checker will likely fail to find the deadlock since the state space becomes to big. Furthermore, if the other threads don't deadlock, then the global system never deadlocks, although the two tasks may. Hence, since the JPF2 model checker currently only looks for global deadlocks, it will never be able to find this local one.

As an experiment, the program was composed with an environment consisting of 40 threads, grouped in pairs, each pair sharing access to an object by updating it (each thread assigns $10,000$ different values to the object). This environment has more than 10^{160} states. When running JPF2 in runtime analysis mode, it prints out 44 messages, one for each time a new locking pattern is recognized (40 of the patterns come from the environment). When these messages no longer get printed, after 25 seconds, one can assume[3] that all patterns have been detected, and by hitting a key on the keyboard, the lock analysis is started. This identifies the original two `Task` threads as being the sinners. The model checker is now launched where only the `Main` thread, and the two `Task` threads are allowed to execute, and the deadlock is found by the model checker in 1.6 seconds. The `Main` thread is included because it starts the `Task` threads, as concluded based on a dependency analysis.

4.2 Algorithm

Most of the work has already been done during runtime analysis. An additional data structure must be introduced, the *race window*, which contains the threads that caused warnings to be issued. Before the model checker is activated, an *extended race window* is calculated, which includes additional threads that may influence the behavior of threads in the original window. The extension is calculated on the basis of a *dependency graph*, created by a dependency analysis also performed during the execution (a third kind of runtime analysis). This extended window is then used in the subsequent model checking by freezing all threads not in the window. That is, the scheduler simply does not schedule threads outside the window.

Figure 11 illustrates the state variables and operations needed to create the window and dependency graph, and the operation for extending the window. The window is just a set of threads. The dependency graph (dgraph) is a mapping from threads t to triples (A, R, W), where A is the ancestor thread that spawned t, R is the set of objects that t reads from, and W is the set of objects that t writes to. Whenever a runtime warning is issued, the 'addWarning' operation is called for each thread involved, adding it to the window. The operations 'startThread', 'readObject', and 'writeObject' update the dependency graph, which after program termination is used by the 'extendWindow' operation to extend

[3] This is a judgment call of course.

the window. The dependency graph is updated when a thread starts another thread with the `start()` method, and when a thread reads from, or writes to a variable in an object. The 'extendWindow' operation performs a fix-point calculation by creating the set of all threads *"reachable"* from the original window by repeatedly including threads that have spawned threads in the window, and by including threads that write to objects that are read by threads in the window. The extended window is used to evaluate whether a thread should be scheduled or not.

```
type Window = setof Thread;
type Dgraph = map from Thread to (Thread × setof Object × setof Object);

Window window;     (* updated when a runtime warning is issued *)
Dgraph dgraph;     (* updated when a thread starts a thread or accesses an object *)

addWarning(Thread thread){
    window = window ∪ {thread}}

startThread(Thread father,Thread son){
    dgraph = dgrap + [son ↦ (father, {}, {})]}

readObject(Thread thread,Object object){
    let (A, R, W) = dgraph(thread){
        dgraph = dgraph + [thread ↦ (A, R ∪ {object}, W)]}}

writeObject(Thread thread,Object object){
    let (A, R, W) = dgraph(thread){
        dgraph = dgraph + [thread ↦ (A, R, W ∪ {object})]}}

Window extendWindow(Window window,Dgraph dgraph){
    Window passed = {};
    Window waiting = window;
    while (waiting ≠ {}){
        get thread from waiting;
        if (thread ∉ passed){
            passed = passed ∪ {thread};
            let (A, R, W) = dgraph(thread){
                if (A ≠ "topmost thread'') waiting = waiting ∪ {A};
                waiting = waiting ∪
                    {thread' | let(_, _, W') = dgraph(thread') in W' ∩ R ≠ {}};
            }
        }
    };
    return passed;}
```

Fig. 11. Operations for creating dependency graph and window.

4.3 Implementation

Two classes, whose interfaces are given below, represent respectively the dependency graph and the race window. The dependency graph can be updated when threads start threads, or access objects. Finally, a method allows to calculate the set of threads reachable from an initial window, based on the dependencies recorded. The race window allows to record threads involved in warnings. Before the model checker is launched the `extendWindow` method will include threads that influence the original window by calling the `reachable` method. The model checker scheduler will finally call the `contains` method whenever it needs to determine whether a particular thread is in the window, in which case it will be allowed to execute.

```
interface iDepend{
  static void    startThread(ThreadInfo father,ThreadInfo son);
  static void    readObject(ThreadInfo th,int objref);
  static void    writeObject(ThreadInfo th,int objref);
  static HashSet reachable(HashSet threads);
}

interface iRaceWindow{
  static void    addWarning(ThreadInfo th);
  static void    extendWindow();
  static boolean contains(String threadName);
}
```

The following bytecodes are instrumented to operate on the dependency graph: INVOKEVIRTUAL for invoking the `start` method on a thread; and PUTFIELD, GETFIELD, PUTSTATIC, GETSTATIC for accessing variables.

5 The RAX Example

In this section we present an example drawn from a real NASA application. The Remote Agent (RA) [24] is an AI-based spacecraft controller programmed in LISP, that has been developed at NASA Ames Research Center. It consists of three components: a Planner that generates plans from mission goals; an Executive that executes the plans; and finally a Recovery system that monitors the RA's status, and suggests recovery actions in case of failures. The Executive contains features of a multi-threaded operating system, and the Planner and Executive exchange messages in an interactive manner. Hence, this system is highly vulnerable to multi-threading errors. In fact, during real flight in May 1999, the RA deadlocked in space, causing the ground crew to put the spacecraft on standby. The ground crew located the error using data from the spacecraft, but asked as a challenge our group if we could locate the error using model checking. This resulted in an effort described in [15], which in turn refers to earlier work on the RA described in [16]. Here we shall give a short account of the error and show how it could have been located with runtime analysis, and furthermore potentially be confirmed using model checking. For this purpose we have modeled the error situation in Java. Note that this Java program represents a small model of part of the RA, as described in [15]. However, although this is not an automated application to a real full-size program, it illustrates the approach.

The major two components to be modeled are events and tasks, as illustrated in Figure 12. The figure shows a Java class `Event` from which event objects can be instantiated. The class has a local counter variable and two synchronized methods, one for waiting on the event and one for signaling the event, releasing all threads having called `wait_for_event`. In order to catch events that occur while tasks are executing, each event has an associated event counter that is increased whenever the event is signaled. A task then only calls `wait_for_event` in case this counter has not changed, hence, there have been no new events since it was last restarted from a call of `wait_for_event`. The figure shows the definition of one of the tasks, the planner. The body of the `run` method contains an infinite loop, where in each iteration a conditional call of `wait_for_event` is executed. The condition is that no new events have arrived, hence the event counter is unchanged.

```
class Event {
  int count = 0;

  public synchronized void wait_for_event() {
    try{wait();}catch(InterruptedException e){};
  }

  public synchronized void signal_event(){
    count = (count + 1) % 3;
    notifyAll();
  }
}

class Planner extends Thread{
  Event event1,event2;
  int count = 0;

  public void run(){
    while(true){
      if (count == event1.count)
        event1.wait_for_event();
      count = event1.count;
      /* Generate plan */
      event2.signal_event();
    }
  }
}
```

Fig. 12. The RAX Error in Java.

To illustrate JPF2's integration of runtime analysis and model checking, the example is made slightly more realistic by adding extra threads as before. The program has 40 threads, each with $10,000$ states, in addition to the Planner and Executive threads, yielding more than 10^{160} states in total. Then we apply JPF2 in its special runtime analysis/model checking mode. It immediately identifies the data race condition using the Eraser algorithm: the variable `count` in class `Event` is accessed unsynchronized by the Planner's run method in the line: "`if (count == event1.count)`", specifically the expression: `event1.count`. This may be enough for a programmer to realize an error, but only if he or she can see the consequences. The JPF2 model checker, on the other hand, can be used to analyze the consequences. Hence, the model checker is launched on a thread window consisting of those threads involved in the data race condition: the Planner and the Executive, locating the deadlock - all within 25 seconds. The error trace shows that the Planner first evaluates the test "`(count == event1.count)`", which evaluates to true; then, before the call of `event1.wait_for_event()` the Executive signals the event, thereby increasing the event counter and notifying all waiting threads, of which there however are none yet. The Planner now unconditionally waits and misses the signal. The solution to this problem is to enclose the conditional wait in a critical section such that no events can occur in between the test and the wait. This error caused the deadlock in the space craft.

6 Conclusions and Future Work

We have presented a new algorithm, the GoodLock algorithm, for detecting deadlock possibilities in programs caused by locks being taken in different orders by parallel running threads. The algorithm is based on an analysis of a single run of the program, and is therefore an example of a runtime analysis algorithm in the same family as the Eraser algorithm which detects data races. The algorithm minimizes false positives by taking account for gate locks that “protect” lock order problems “further down”. An interesting observation is that a Java program with everything stripped away except the taking and releasing of locks may still have a state space that is too large to model check. The GoodLock algorithm can even in this case be superior to a model checking of such a synchronization skeleton. The algorithm is based on a post-execution analysis in contrast to the Eraser algorithm which performs an on-the-fly analysis. We have furthermore suggested how to use the results of a runtime analysis to guide a model checker for their mutual benefit: the warnings yielded by the runtime analysis can help focus the search of the model checker, which in turn can help eliminate false positives generated by the runtime analysis, or generate an error trace showing how the warnings can manifest itself in an error. In order to create the smallest possible self-contained sub-program to be model checked based on warnings from the runtime analysis, a runtime dependency analysis is introduced, which very simply records dependencies between threads and objects. In addition to implementing all of the above mentioned techniques, we have implemented the existing generic Eraser algorithm to work for Java by instrumenting bytecodes. This is according to one of the authors of [26] amongst the first attempts outside SRC to do this.

Future work will consist of improving the Eraser algorithm to give less false positives, in particular in the context of initializations of objects. The GoodLock algorithm will also be generalized to deal with deadlocks between multiple threads. One can furthermore consider alternative kinds of runtime analysis, for example analyzing issues concerned with the use of the built-in `wait` and `notify` thread methods in Java. A runtime analysis typically cannot guarantee that a program property is satisfied since only a single run is examined. The results, however, are often pretty accurate because the chosen run does not itself have to violate the property, in order for the property's potential violation in other runs to be detected. In order to achieve even higher assurance, one can of course consider activating runtime analysis *during* model checking (rather than before as described in this paper), and we intend to make that experiment. Note that it will not be necessary to explore the entire state space in order for this simultaneous combination of runtime analysis and model checking to be useful. Even though runtime analysis scales relatively well, it also suffers from memory problems when analyzing large programs. Various optimizations of data structures used to record runtime analysis information can be considered, for example the memory optimizations suggested in [26]. One can furthermore consider only doing runtime analysis on objects that are really shared by first determining the sharing structure of the program. This in turn can be done using runtime analysis, or some form of static analysis. Of course, at the extreme the runtime analysis can be performed on a separate computer. We intend to investigate how the runtime analysis information can be used to feed a program slicer [14], as an alternative to the runtime dependency analysis described in this paper.

References

1. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muoz, S. Owre, H. Rue, J. Rushby, V. Rusu, H. Sadi, N. Shankar, E. Singerman, and A. Tiwari. An Overview of SAL. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, June 2000.
2. S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *CAV'98: Computer-Aided Verification*, number 1427 in LNCS, 1998.
3. D. L. Bruening. Systematic Testing of Multithreaded Java Programs. Master's thesis, MIT, 1999.
4. T. Cattel. Modeling and Verification of sC++ Applications. In *Proceedings of TACAS98: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *LNCS*, LISBON, April 1998.
5. J. Corbett. Constructing Compact Models of Concurrent Java Programs. In *Proceedings of the ACM Sigsoft Symposium on Software Testing and Analysis*, March 1998. Clearwater Beach, Florida.
6. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.
7. P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 4(2):511–547, August 1992.
8. S. Das, D. Dill, and S. Park. Experience with Predicate Abstraction. In *CAV '99: 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, 1999.
9. C. Demartini, R. Iosif, and R. Sist. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.
10. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, California, USA, 1998.
11. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.
12. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

13. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *CAV '97: 6th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, 1997.
14. J. Hatcliff, J.C. Corbett, M.B. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Proc. of the 1999 Int. Symposium on Static Analysis*, 1999.
15. K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal Analysis of the Remote Agent Before and After Flight. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, June 2000.
16. K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proceedings of the 4th SPIN workshop, Paris, France*, November 1998. To appear in IEEE Transactions of Software Engineering.
17. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.
18. K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In M-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *LNCS*, pages 662–681. Springer-Verlag, 1996. An experiment in program abstraction.
19. K. Havelund and J. Skakkebaek. Applying Model Checking in Java Verification. In *Proceedings of the 7th Workshop on the SPIN Verification System*, volume 1680 of *LNCS*, Toulouse, France., September 1999.
20. G. Holzmann and M. Smith. A Practical Method for Verifying Event-Driven Software. In *Proc. ICSE99, International Conference on Software Engineering, Los Angeles*. IEEE/ACM, May 1999.
21. G.J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
22. R. Iosif, C. Demartini, and R. Sisto. Modeling and Validation of JAVA Multithreaded Applications using SPIN. In *Proceedings of the Fourth Workshop on the SPIN Verification System*, Paris, November 1998.
23. JavaClass. http://www.inf.fu-berlin.de/ dahm/JavaClass/+.
24. N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.
25. D. Park, U. Stern, and D. Dill. Java Model Checking. In *Proc. of the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, June 2000.
26. S. Savage, M. Burrows, G. Nelson, and P. Sobalvarro. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
27. W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - Second Generation of a Java Model Checker. In *Proc. of Post-CAV Workshop on Advances in Verification, Chicago*, July 2000.
28. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proc. of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.
29. W. Visser, S. Park, and J. Penix. Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking. Submitted for publication.