1-1-2013

# Similarity-based search for model checking: a pilot study with java pathfinder

Elmin Ibrahimov
*University of Wollongong*, ei978@uowmail.edu.au

Jixing Wang
*University of Wollongong*

Zhiquan Zhou
*University of Wollongong*, zhiquan@uow.edu.au

# Similarity-based search for model checking: a pilot study with java pathfinder

## Abstract

When a model checker cannot explore the entire state space because of limited resources, model checking becomes a kind of testing with an attempt to find a failure (violation of properties) quickly. We consider two state sequences in model checking: (i) the sequence in which new states are generated, and (ii) the sequence in which the states generated in sequence (i) are checked for property violation. We observe that neighboring states in sequence (i) often have similarities in certain ways. Based on this observation we propose a search strategy, which generates sequence (ii) in such a way that similar states are evenly spread over the sequence. As a result, neighboring states in sequence (ii) can have a higher diversity. A pilot empirical study with Java PathFinder suggests that the proposed strategy can outperform random search in terms of creating equal or smaller number of states to detect a failure.

## Keywords

java, pathfinder, pilot, study, checking, similarity, model, search

## Disciplines

Engineering | Science and Technology Studies

## Publication Details

Ibrahimov, E., Wang, J. & Zhou, Z. (2013). Similarity-based search for model checking: a pilot study with java pathfinder. 13th International Conference on Quality Software (pp. 238-244). IEEE Xplore: IEEE.

# Similarity-Based Search for Model Checking: A Pilot Study with Java PathFinder

Elmin Ibrahimov Jixing Wang Zhi Quan Zhou *

School of Computer Science and Software Engineering

University of Wollongong

Wollongong, NSW 2522, Australia

*Abstract* —— **When a model checker cannot explore the entire state space because of limited resources, model checking becomes a kind of testing with an attempt to find a failure (violation of properties) quickly. We consider two state sequences in model checking: (i) the sequence in which new states are generated, and (ii) the sequence in which the states generated in sequence (i) are checked for property violation. We observe that neighboring states in sequence (i) often have similarities in certain ways. Based on this observation we propose a search strategy, which generates sequence (ii) in such a way that similar states are evenly spread over the sequence. As a result, neighboring states in sequence (ii) can have a higher diversity. A pilot empirical study with Java PathFinder suggests that the proposed strategy can outperform random search in terms of creating equal or smaller number of states to detect a failure.**

**Keywords:** Model checking; random search; similarity-based search; heuristics; adaptive random sequence; Java PathFinder.

## I. Introduction

A main challenge in model checking is the state space explosion problem. To alleviate this problem, different strategies can be used to guide the search in the state space, aiming at detecting a failure early before resources are exhausted [1]. In this situation, model checking becomes a kind of testing.

In the process of model checking, two sequences of states can be identified: (*i*) the sequence where new states are generated, and (*ii*) the sequence where the states generated in sequence (*i*) are checked/verified. Note that the activities of state generation and state verification can be interleaved. This is because, when verifying a state, its new child states may be derived. In breadth-first search (BFS), sequence (*ii*) can be the same as sequence (*i*): states generated earlier are also verified earlier. Whereas in depth-first search (DFS), the model checker explores as far as possible along each branch before backtracking, and newly generated states can be verified before previously generated states. In DFS, therefore, sequence (*ii*) is normally different from sequence (*i*).

In random search, the next state to verify is a state randomly chosen from sequence (*i*). Random search has a unique advantage over BFS and DFS: BFS can easily run out of memory; and DFS's effectiveness depends on the location of the error state in the search tree. Random search does not have these problems associated with the deterministic search algorithms, and can be considered the simplest and the most basic search algorithm/heuristic.

The aim of this research is to improve the effectiveness of random search. We propose a *similarity-based search* (SBS) strategy, which is a variant of random search. The rest of this paper is organized as follows: Section II explains the basic idea of SBS and states our research question; Section III reviews related work in the field of software testing; Section IV briefly introduces Java PathFinder (JPF), the software model checker used in the empirical study of this research. Section V presents an algorithm that provides service to SBS. Section VI presents a pilot empirical study with 8 Java programs, where random search and SBS are compared. Section VII discusses the validity of this work and concludes the paper.

## II. The Basic Idea Of Similarity-Based Search

Consider state sequences (*i*) and (*ii*) introduced in Section (I). We observe that neighboring states in sequence (*i*) often have similarities in certain ways. This is because, in the process of state generation, closely related states are often generated in rapid sequence. For example, child states of the same parent state (that is, branches expanded from the same node) are often identified/generated together and, hence, they become neighboring states in sequence (*i*). These child states are "similar" in that they have the same ancestors. It is our intuition that error states tend to cluster. For example, let *a*, *b* and *c* be 3 states, where *a* and *b* are similar (e.g. they have the same parent node), and *c* is different from *a* (e.g. *c* is located far away from *a* in the search tree). If *a* is an error state, then, intuitively, the chance of *b* also being an error state should be higher than that of *c*, as *b* is more similar to *a*. Consequently, non-error states should also cluster. Therefore, if previously checked states are all correct, the next state to check should be far apart from the previously checked states. In other words, similar (neighboring) states in sequence (*i*) should be evenly spread across sequence (*ii*).

It is this concept of evenly spreading similar states that forms the basic intuition of our similarity-based search (SBS) strategy for model checking. Based on this concept many different algorithms can be developed to achieve the even spread of states. Note that pure random selection may not

* All correspondence should be addressed to Dr. Zhi Quan Zhou, School of Computer Science and Software Engineering, University of Wollongong, Wollongong, NSW 2522, Australia. Email: zhiquan@uow.edu.au. Telephone: (61-2) 4221 5399.

achieve this goal because there is always a chance to (randomly) select a state that is close to some of the previously checked states. Our research question is stated below:

*Can SBS be effectively implemented in such a way that it outperforms random search?*

Before addressing the above research question, we first review some related work in the next section.

## III. Related Work

In the field of software testing, there is a family of test case generation methods, known as *adaptive random testing* (ART), which is designed to improve the fault-detection effectiveness of random testing by enforcing an even spread of randomly generated test cases over the input domain [2,3]. ART is based on the observation that failure-causing inputs tend to cluster, forming contiguous failure regions.

Various ART algorithms have been developed, with different orders of time complexity, ranging from $O(n^2)$ to $O(n)$, where n is the number of test cases generated. ART well preserves the randomness of test cases, and outperforms RT in terms of both F-measure and P-measure [2]. Naturally, however, generating an adaptive random test case using the location information of already executed test cases do require more computational overhead as compared with the generation of a pure random number.

In SBS for model checking, when a new state is generated, it will be assigned an (adaptive random) priority number, and the descending order of the priority numbers will decide the order in which the states are verified. This method will be explained shortly.

## IV. A Brief Introduction to Java PathFinder

Essentially, Java PathFinder (JPF) is an explicit state software model checker for Java bytecode [4,5]. The JPF core is a Java virtual machine (JVM); but different from a normal JVM, JPF can run Java bytecode more than once, "theoretically in all possible ways, checking for property violations like deadlocks or unhandled exceptions along all potential execution paths." When a violation of the property is found, JPF will stop to report the entire execution trace that leads to the violation (failure). JPF was developed by the NASA Ames Research Center, and was open sourced in 2005.

The algorithms that JPF uses to search for property violations in the state space are configurable and extensible. In this research we create our own search algorithm, which is an implementation of SBS, and will compare its effectiveness against that of random search with respect to the number of new states created (the lower the better).

More specifically, JPF has two top level modules, namely, the JVM and the Search objects. The latter are responsible for "selecting the state from which the JVM should proceed, either by directing the JVM to generate the next state (forward), or by telling it to backtrack to a previously generated one" [5].

The program under model checking (Java bytecode) is loaded and run by the JVM and driven by a prescribed search algorithm. The main Search implementations "include a simple depth-first search (DFSearch), and a priority-queue based search that can be parameterized to do various search types" [5]. In the priority-queue based search, JPF assigns an integer-valued priority number to each new state. It is always the state that has the largest priority number that is checked first for property violations. In random search, for instance, each new state is given a random priority number. Therefore, states are checked for property violations following a random sequence.

Our SBS algorithm differs from random search in that, when a new state is generated, it is an adaptive random integer (rather than random integer) that is assigned to the new state to serve as its priority number. As a result, states are checked against property violations by following an adaptive random sequence. Compared with a random sequence, the adaptive random sequence can more evenly spread similar states (that is, neighboring states in sequence (*i*)) and, therefore, neighboring states in sequence (*ii*) will have less similarity, or higher diversity (where sequences (*i*) and (*ii*) are defined in Section I).

## V. An Algorithm of Generating Priority Numbers

Let $S=(s_1, s_2, \ldots, s_n)$ be sequence (*i*) as defined in Section I, that is, S is a sequence of n consecutively generated states; let $T=(t_1, t_2, \ldots, t_n)$ be an adaptive random sequence of integers in the range of [0, MAX], where MAX is the maximum integer of the system and MAX > n. We say $t_i$ is an adaptive random number, $i = 1, 2, \ldots, n$. Most of the ART family of algorithms [2,3] can be used by SBS to generate T, and SBS sets the priority number of state $s_i$ to $t_i$, $i=1, 2, \ldots, n$. When the next state $s_{n+1}$ is generated, SBS will call the adaptive random number generator to generate the next adaptive random number $t_{n+1}$, which will serve as the priority number of $s_{n+1}$. In this way, the generated states are checked for property violations following the descending order of their priority numbers, and state generation and state verification can be interleaved.

There are notable differences between ART and our approach. ART generates/selects test cases from the input domain. In most ART algorithms, when a test case is being generated, its ordinal rank in the test case execution sequence is known because the test case generation sequence and the test case execution sequence in most ART algorithms are the same. In our approach, for a given state $s_i$, we generate its priority number, and the priority number implies the ordinal rank of $s_i$ in the verification sequence; further, this ordinal rank can be changed in the future because a future new state may receive a priority number larger than that of $s_i$.

The connection between our approach and ART is that our SBS module uses an ART algorithm to generate a priority number (over the one-dimensional integer domain) for each new state. In this paper, we consider but one of the ART family of algorithms, namely, Iterative Partitioning ART (IP-ART) [6]. The original IP-ART algorithm works on the real domain, and we revised the algorithm so that it works on the integer domain. The structure of the algorithm is briefly described below:

```
Begin AdaptiveRandomSequenceGenerator
1.  Initialize **CandidateSet** to {[0, MAX]};
    /* MAX is the maximum integer of the system. It
    is a precondition that MAX is greater than the
    total number of states. */
2.  While(stopping criterion is not met)
3.   If (**CandidateSet** is not empty)
4.     Randomly select an integer **i**
         from **CandidateSet**;
5.     Set an exclusion zone surrounding **i**;
6.     Update **CandidateSet** accordingly;
7.   Else
8.     Reduce the radius of exclusion zones;
9.     Update **CandidateSet** accordingly;
10.  EndIf
11. EndWhile
End of AdaptiveRandomSequenceGenerator
```

The algorithm AdaptiveRandomSequenceGenerator generates an adaptive random sequence of integers in the range [0, MAX], where MAX is large enough. In our implementation, MAX is set to the largest integer of the system. The variable CandidateSet stores available areas where the next adaptive random number can be generated from. Statements 3 to 6 mean that if there are available areas then an integer is randomly generated (selected) from these areas. After that an exclusion zone is created around the selected number so that any point in the exclusion zone will not be selected in the future until available areas are used up – in this situation the radius of the exclusion zones will be reduced to create some available areas, as shown in statements 8 and 9. For ease of presentation and understanding, the above algorithm shows the entire process of generating a sequence of adaptive random integers; in our actual implementation, every time a new state is generated, the SBS module will run the "While" loop (statements 2 to 11) to generate only one adaptive random integer as the priority number of the new state. In other words, statement 4 is executed once and only once for each new state.

The algorithm AdaptiveRandomSequenceGenerator is best explained using an easy-to-understand example. The value range of the priority numbers is [0, MAX]. We maintain two sets, namely, the Point Set, which records the points already generated (initialized to be empty), and the Candidate Set, which is a set of ranges from which a new value can be generated (initialized to be {[0, MAX]}). A variable, Distance, is initialized to MAX/2. An integer value, P1, is randomly generated from the range indicated by the Candidate Set. The variable Distance gives the radius of the exclusion zone, that is, Distance is the minimum difference between two generated values. Therefore, any value whose distance to P1 is within MAX/2 will be excluded from consideration next time. Figure 1 shows the initialization of variables and generation of P1.

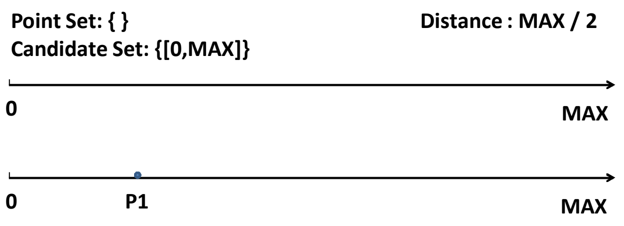

**Figure 1** Initialization of variables (upper) and generation of the first integer value, P1 (lower).

Then P1 is put into the Point Set; and Candidate Set is reduced to the remaining area that is not covered by the exclusion zone, namely, Candidate Set = {[N1, MAX]}, where N1 is the boarder of the exclusion zone. This is illustrated in Figure 2.

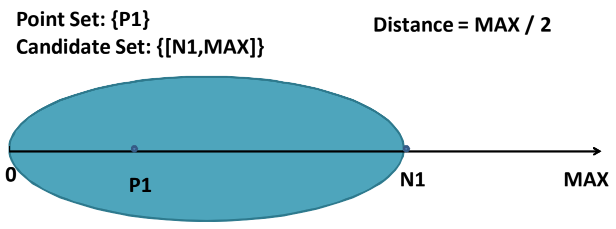

**Figure 2** The exclusion zone is [0, N1), and the next value will be generated from the area [N1, MAX].

Figure 3 (upper) shows that the next value will be directly generated from the range [N1, MAX]. Note that this method is different from R-ART (that is, ART by Exclusion): the latter generates random numbers one by one until one number falls outside of all exclusion zones, and that number is then selected; whereas our IP-ART algorithm is based on partitioning: a random number from the Candidate Set is directly generated without trial and error.

Figure 3 (lower) shows that a new value, P2, is generated, and an exclusion zone surrounding P2 is created. Now the entire area [0, Max] is covered by the exclusion zones of P1 and P2, and no new value can be generated as Candidate Set becomes empty. At this time the value of Distance will be reduced using a certain rate (our implementation used a deduction rate of 10%). When Candidate Set becomes non-empty (see Figure 4), we randomly select a value from the regions included in Candidate Set. This procedure is repeated until the stopping criterion is met (such a criterion can be, for example, detection of a property violation or exhaustion of resources). Observed curvatures of the time cost of this algorithm are shown in Figure 5.

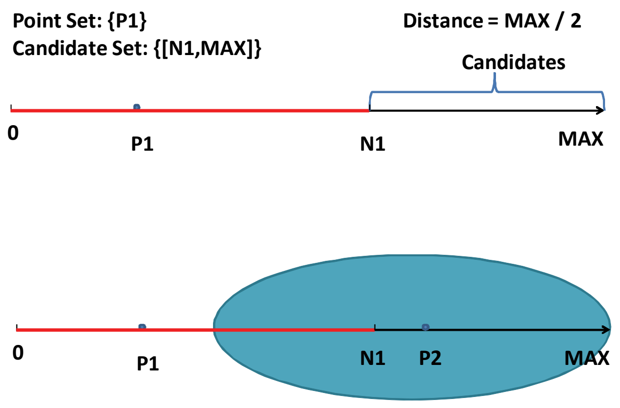

**Figure 3** Generating the second integer value, P2, directly from the Candidate Set (upper), and calculating the exclusion zone of P2 (lower).

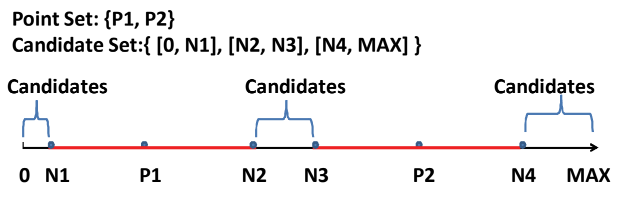

**Figure 4** Decreasing the value of Distance to allow Candidate Set to contain regions where the next value can be generated.

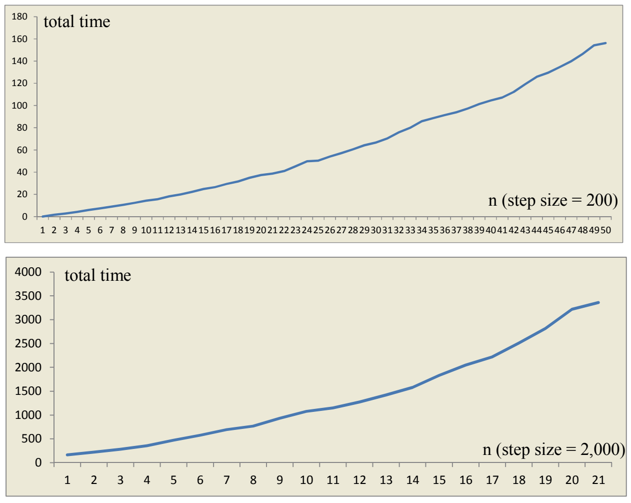

**Figure 5** Experimental results of total time (average of 1,000 trials, in milliseconds) consumed for generating an AR sequence of n priority numbers, where n varied from 1 to 9,801 with step size of 200 (upper) and from 10,000 to 50,000 with step size of 2,000 (lower).

## VI. A Pilot Empirical Study

A pilot empirical study has been conducted by applying the random search and SBS to a total of 8 small Java programs. Some of these programs are taken from the JPF package, and property violations being checked include deadlock, uncaught exceptions, and assertion errors. JPF will stop when the first violation of a property is detected. These 8 programs are named AddNumber, BankingDeadlock, Bow, Crossing, DiningPhil, ReadWrite, Resources, and TwoWays. Their source code is listed in Appendix.

The experimental results are given in Table 1. The results show that the number of new states generated by SBS is equal to or smaller than that of random search for every subject program. More specifically, both random search and SBS gave "1" for AddNumber and ReadWrite (which are trivial), and both gave "21" for Resources. For all other 5 programs, SBS outperformed random search in terms of creating fewer states to detect the first property violation. In average, the number of states created by SBS is 91.03% that of random heuristic.

**TABLE 1** Experimental results (average numbers of new states created to detect a property violation, out of 100 trials for each subject program).

| Program | Random search | SBS | SBS / Random |
|---|---|---|---|
| AddNumber | 1 | 1 | 100.00% |
| BankingDeadlock | 22 | 20 | 90.91% |
| Bow | 18 | 16 | 88.89% |
| Crossing | 634 | 538 | 84.86% |
| DiningPhil | 1591 | 1489 | 93.59% |
| ReadWrite | 1 | 1 | 100.00% |
| Resources | 21 | 21 | 100.00% |
| TwoWays | 30 | 29 | 96.67% |
| Average | 290 | 264 | 91.03% |

## VII. Discussions and Conclusion

When a model checker cannot traverse the entire state space because of limited resources, model checking becomes a kind of testing with an attempt to detect a property violation quickly before resources are exhausted. In this paper we proposed a novel search strategy, namely, SBS, to improve the effectiveness of random search. SBS is a variant of random search: it preserves the randomness of search and does not have the problems associated with the deterministic BFS and DFS strategies. A small-scale empirical evaluation with JPF model checker suggests that SBS outperformed random search in terms of creating fewer states to detect the first property violation, hence providing a positive answer to the research question raised in Section II.

The main threat to validity of this work is the small scale of experiments. Only 8 small Java programs were used for the empirical study. Further empirical studies are obviously necessary. Furthermore, more efficient algorithms of generating adaptive random sequences in the one-dimensional integer domain exist and they should be investigated in future research.

Our method is based on the observation that states generated in rapid sequence during model checking often have similarities in certain ways. However, we do not assume an explicit ordering of the candidate states on which the selection is made (such as breadth-first or depth-first search sequences). Nevertheless, we do assume that error states cluster. Investigation into the validity of this assumption is worth an in-depth study in the future.

## VIII. ACKNOWLEDGMENTS

This research was supported in part by a linkage grant of the Australian Research Council (project ID: LP100200208).

## REFERENCES

[1] A. Groce and W. Visser, “Heuristics for model checking Java programs,” International Journal on Software Tools for Technology Transfer, vol. 6, pp. 260—276, 2004.

[2] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, “An orchestrated survey on automated software test case generation,” A. Bertolino, J. J. Li and H. Zhu (Editor/Orchestrators), Journal of Systems and Software, to appear.

[3] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, “Adaptive random testing: The ART of test case diversity,” Journal of Systems and Software, vol. 83, no. 1, pp. 60–66, 2010.

[4] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, “Model checking programs,” Automated Software Engineering Journal, vol. 10, no. 2, 2003.

[5] Java PathFinder Home Page. http://babelfish.arc.nasa.gov/trac/jpf.

[6] T. Y. Chen, D. H. Huang, and Z. Q. Zhou, “On adaptive random testing through iterative partitioning,” Journal of Information Science and Engineering, vol. 27, no. 4, pp. 1449–1472, 2011.

**Appendix:** Source Code of the 8 Subject Java Programs

**AddNumber.java**

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import gov.nasa.jpf.jvm.Verify;
public class AddNumber {
   public static void main(String[] args) throws InterruptedException {
      Verify.beginAtomic();
      NumberContainer a = new NumberContainer();
      NumberContainer b = new NumberContainer();
      Thread tab = new Thread(new RunnableDeadlock(a, b, "AB"));
      Thread tba = new Thread(new RunnableDeadlock(b, a, "BA"));
      tab.start();
      tba.start();
      System.out.println("M 1");
      tab.join();
      tba.join();
      System.out.println("M 2");
      System.out.println("A: ");
      a.print();
      System.out.println("B: ");
      b.print();
      Verify.endAtomic();
   }
}
class NumberContainer {
   private List<Number> elements = new ArrayList<Number>();
   public void add(Number number) {
      elements.add(number);
   }
   public void print() {
      System.out.println(elements);
   }
}
class RunnableDeadlock implements Runnable {
   private List<Integer> numbers = new ArrayList<Integer>();
   private NumberContainer n1;
   private NumberContainer n2;
   private String name;
   public RunnableDeadlock(NumberContainer n1, NumberContainer n2,
String name) {
      super();
      this.n1 = n1;
      this.n2 = n2;
      this.name = name;
   }
   @Override
   public void run() {
      System.out.println("Lock n1 " + name);
      synchronized (n1) {
         doHeavyWork();
         System.out.println("Lock n2 " + name);
         synchronized (n2) {
            n1.add(numbers.get(0));
            n2.add(numbers.get(numbers.size() - 1));
         }
      }
   }
   public void doHeavyWork() {
      long start = System.currentTimeMillis();
      System.out.println("start heavy work");
      Random random = new Random();
      for (int i = 0; i < 100000; i++) {
         numbers.add(random.nextInt());
      }
      Collections.sort(numbers);
      long end = System.currentTimeMillis();
      System.out.println("end heavy work: " + (end - start));
   }
}
```

**BankingDeadlock.java**

```
import gov.nasa.jpf.jvm.Verify;
public class BankingDeadlock {
   public static void main(String args[]) {
      Verify.beginAtomic();
      Account accOne = new Account(100);
      Account accTwo = new Account(200);
      Thread john = new Thread(new Clerk("John", 60, accOne, accTwo));
      Thread jim = new Thread(new Clerk("Jim", 30, accTwo, accOne));
      john.start();
      jim.start();
      Verify.endAtomic();
   }
}
class Account {
   int balance;
   public Account(int balance) {
      this.balance = balance;
   }
   public int getBalance(Clerk c) {
    System.out.format("%s gets the balance:
Current balance is %d%n", c.name, balance);
    return balance;
  }
  public void setBalance(Clerk c, int
newBalance) {
    System.out.format("%s sets the balance: Old
balance is %d%n",
        c.name, balance);
    balance = newBalance;
    System.out.format("%s sets the balance:
New balance is %d%n", c.name, balance);
  }
}
class Clerk implements Runnable {
  String name;
  int amount;
  Account acc;
  Account otherAcc;
  public Clerk(String name, int amount, Account
acc, Account otherAcc) {
    this.name = name;
    this.amount = amount;
    this.acc = acc;
    this.otherAcc = otherAcc;
  }
  public void run() {
    int balance, newBalance;
    synchronized (acc) {
      transfer(acc, otherAcc, amount);
    }
  }
  public void transfer(Account from, Account to,
int ammount) {
    synchronized (from) {
      int from_balance = from.getBalance(this);
      int from_balance_new = from_balance -
amount;
      from.setBalance(this, from_balance_new);
    }
    synchronized (to) {
      int to_balance = to.getBalance(this);
      int to_balance_new = to_balance +
amount;
      from.setBalance(this, to_balance_new);
    }}}
```

**Bow.java**

```
import gov.nasa.jpf.jvm.Verify;
public class Bow {
  static class Friend {
    private final String name;
    public Friend(String name) {
      this.name = name;
    }
    public String getName() {
      return this.name;
    }
    public synchronized void bow(Friend bower)
{
      System.out.format("%s: %s has bowed to
me!%n",
          this.name, bower.getName());
      bower.bowBack(this);
    }
    public synchronized void bowBack(Friend
bower) {
      System.out.format("%s: %s has bowed
back to me!%n",
          this.name, bower.getName());
    }}
  public static void main(String[] args) {
    Verify.beginAtomic();
    final Friend alphonse = new
Friend("Alphonse");
    final Friend gaston = new Friend("Gaston");
    new Thread(new Runnable() {
      public void run() {
        alphonse.bow(gaston);
      }}).start();
    new Thread(new Runnable() {
      public void run() {
        gaston.bow(alphonse);
      }}).start();
    Verify.endAtomic();
  }}
```

**Crossing.java**

```
import gov.nasa.jpf.jvm.Verify;
class Constants {
  public static final boolean east = true;
  public static final boolean west = false;
}
class Torch {
  public static boolean side = Constants.east;
  public String toString() {
    if (side == Constants.east) {
      return "east";
    } else {
      return "west";
    }}}
class Bridge {
  static Person[] onBridge = new Person[2];
  static int numOnBridge = 0;
  public static boolean isFull() {
    return numOnBridge != 0;
  }
  public static int Cross() {
    int time = 0;
    Torch.side = !Torch.side;
    if (numOnBridge == 1) {
      onBridge[0].side = Torch.side;
      time = onBridge[0].time;
    } else {
      assert onBridge[0] != null : "Argh, null " +
numOnBridge;
      assert onBridge[1] != null;
      onBridge[0].side = Torch.side;
      onBridge[1].side = Torch.side;
      if (onBridge[0].time > onBridge[1].time) {
        time = onBridge[0].time;
      } else {
        time = onBridge[1].time;
      }}
    return time;
  }
  public static void clearBridge() {
    if (numOnBridge == 0) {
      return;
    } else if (numOnBridge == 1) {
      onBridge[0] = null;
      numOnBridge = 0;
    } else {
      onBridge[0] = null;
      onBridge[1] = null;
      numOnBridge = 0;
    }}
  public static void initBridge() {
    onBridge[0] = null;
    onBridge[1] = null;
    numOnBridge = 0;
  }
  public static boolean tryToCross(Person th) {
    if ((numOnBridge < 2) && (onBridge[0] !=
th) && (onBridge[1] != th)) {
      onBridge[numOnBridge++] = th;
      return true;
    } else {
      return false;
    }}}
class Person {
  int id;
  public int time;
  public boolean side;
  public Person(int i, int t) {
    time = t;
    side = Constants.east;
    id = i;
  }
  public void move() {
    if (side == Torch.side) {
      if (!Verify.getBoolean()) {
        Bridge.tryToCross(this);
      }}}
  public String toString() {
    return "" + id;
  }}
public class Crossing {
  public static native void setTotal(int time);
  public static native int getTotal();
  public static void main(String[] args) {
    boolean isNative = false;
    if (isNative) {
      setTotal(30);
    }
    int total = 0;
    boolean finished = false;
    Bridge.initBridge();
    Person p1 = new Person(1, 1);
    Person p2 = new Person(2, 2);
    Person p3 = new Person(3, 5);
    Person p4 = new Person(4, 10);
    while (!finished) {
      p1.move();
      p2.move();
      p3.move();
      p4.move();
      if (Bridge.isFull()) {
        total += Bridge.Cross();
        if (isNative) {
          Verify.ignoreIf(total > getTotal());
        } else {
          Verify.ignoreIf(total > 17);
        }
        Bridge.clearBridge();
        finished = !(p1.side || p2.side || p3.side ||
p4.side);
      }
    }
    if (isNative) {
      if (total < getTotal()) {
        System.out.println("new total " + total);
        setTotal(total);
        assert (total > getTotal());
      }
    } else {
      System.out.println("total time = " + total);
      assert (total > 17);
    }
  }
  static void printConfig(Person p1, Person p2,
Person p3, Person p4,
      int total) {
    if (p1.side == Constants.east) {
      System.out.print("p1(" + p1.time + ")");
```

```
        }
        if (p2.side == Constants.east) {
            System.out.print("p2(" + p2.time + ")");
        }
        if (p3.side == Constants.east) {
            System.out.print("p3(" + p3.time + ")");
        }
        if (p4.side == Constants.east) {
            System.out.print("p4(" + p4.time + ")");
        }
        System.out.print(" - " + total + " -> ");
        if (p1.side == Constants.west) {
            System.out.print("p1(" + p1.time + ")");
        }
        if (p2.side == Constants.west) {
            System.out.print("p2(" + p2.time + ")");
        }
        if (p3.side == Constants.west) {
            System.out.print("p3(" + p3.time + ")");
        }
        if (p4.side == Constants.west) {
            System.out.print("p4(" + p4.time + ")");
        }
        System.out.println();
    }}
```

**DiningPhil.java**

```
import gov.nasa.jpf.jvm.Verify;
public class DiningPhil {
    static class Fork {
    }
    static class Philosopher extends Thread {
        Fork left;
        Fork right;
        public Philosopher(Fork left, Fork right) {
            this.left = left;
            this.right = right;
            start();
        }
        public void run() {
            synchronized (left) {
                synchronized (right) {
                }}}}
    static final int N = 6;
    public static void main(String[] args) {
        Verify.beginAtomic();
        Fork[] forks = new Fork[N];
        for (int i = 0; i < N; i++) {
            forks[i] = new Fork();
        }
        for (int i = 0; i < N; i++) {
            new Philosopher(forks[i], forks[(i + 1) %
N]); }
        Verify.endAtomic();
    }}
```

**ReadWrite.java**

```
import java.util.concurrent.locks.*;
import java.lang.management.*;
import gov.nasa.jpf.jvm.Verify;
public class ReadWrite {
    static ReentrantReadWriteLock lock = new
ReentrantReadWriteLock();
    public static void main(String[] args) throws
Exception {
        Verify.beginAtomic();
        Reader reader = new Reader();
        Writer writer = new Writer();
        sleep(10);
        System.out.println("finding deadlocked
threads");
        ThreadMXBean tmx =
ManagementFactory.getThreadMXBean();
        long[] ids = tmx.findDeadlockedThreads();
        if (ids != null) {
            ThreadInfo[] infos =
tmx.getThreadInfo(ids, true, true);
            System.out.println("the following threads
are deadlocked:");
            for (ThreadInfo ti : infos) {
                System.out.println(ti);
            }}
        System.out.println("finished finding
deadlocked threads");
        Verify.endAtomic();
    }
    static void sleep(int seconds) {
        try {
            Thread.currentThread().sleep(seconds *
1000);
        } catch (InterruptedException e) {
        }}
    static class Reader implements Runnable {
        Reader() {
            new Thread(this).start();
        }
        public void run() {
            sleep(2);
            System.out.println("reader thread getting
lock");
            lock.readLock().lock();
            System.out.println("reader thread got
lock");
            synchronized (lock) {
                System.out.println("reader thread inside
monitor!");
                lock.readLock().unlock();
            }}}
    static class Writer implements Runnable {
        Writer() {
            new Thread(this).start();
        }
        public void run() {
            synchronized (lock) {
                sleep(4);
                System.out.println("writer thread
getting lock");
                lock.writeLock().lock();
                System.out.println("writer thread got
lock!");
            }}}}
```

**Resources.java**

```
import gov.nasa.jpf.jvm.Verify;
public class Resources {
    public static void main(String[] args) {
        Verify.beginAtomic();
        final Object resource1 = "resource1";
        final Object resource2 = "resource2";
        Thread t1 = new Thread() {
            public void run() {
                synchronized (resource1) {
                    System.out.println("Thread 1: locked
resource 1");
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                    }
                    synchronized (resource2) {
                        System.out.println("Thread 1:
locked resource 2");
                    }
                }
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                synchronized (resource2) {
                    System.out.println("Thread 2: locked
resource 2");
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                    }
                    synchronized (resource1) {
                        System.out.println("Thread 2:
locked resource 1");
                    }
                }
            }
        };
        t1.start();
        t2.start();
        Verify.endAtomic();
    }
}
```

**TwoWays.java**

```
import gov.nasa.jpf.jvm.Verify;
public class TwoWays implements Runnable {
    private String o1 = "lock1";
    private String o2 = "lock2";
    private String waysName;
    public TwoWays(String waysName) {
        super();
        this.waysName = waysName;
    }
    @Override
    public void run() {
        if (waysName.equals("way1")) {
            synchronized (o1) {
                try {
                    System.out.println("Lock o1");
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (o2) {
                    System.out.println("way1:Lock o1
o2");
                }
            }
        } else if (waysName.equals("way2")) {
            synchronized (o2) {
                try {
                    System.out.println("Lock o2");
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (o1) {
                    System.out.println("way2:Lock o2
o1");
                }}}}
    public static void main(String[] args) {
        Verify.beginAtomic();
        Thread t1 = new Thread(new
TwoWays("way1"));
        Thread t2 = new Thread(new
TwoWays("way2"));
        t1.start();
        t2.start();
        Verify.endAtomic();
    }}
```