Java PathFinder User Guide

Klaus Havelund

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

August 3, 1999

Abstract

The Java PathFinder, JPF, is a translator from a subset of Java 1.0 to PROMELA, the programming language of the SPIN model checker. The purpose of JPF is to establish a framework for verification and debugging of Java programs based on model checking. The main goal is to automate program verification such that a programmer can apply it in the daily work without the need for a specialist to manually reformulate a program into a different notation in order to analyze the program. The system is especially suited for analyzing multi-threaded Java applications, where normal testing usually falls short. The system can find deadlocks and violations of boolean assertions stated by the programmer in a special assertion language. This document explains how to use JPF.
## Contents

1 Introduction ........................................... 4

2 Specifications ........................................ 5
   2.1 Assertion Methods .................................. 5
   2.2 Error Method ....................................... 5
   2.3 Print Methods ...................................... 6
   2.4 Atomic Execution Methods ......................... 6
   2.5 Non-Deterministic Switch Statements .............. 7

3 Verification Parameters in the Code .................... 8

4 A Program Example: The Bounded Buffer ................. 9
   4.1 The Buffer Class .................................. 9
   4.2 Setting up an Environment .......................... 11
   4.3 Property Specification ............................. 12
   4.4 Predicting Number of Objects Created ............ 12

5 A Guided Tour .......................................... 14
   5.1 Compiling, Executing and Translating the Program ... 14
   5.2 Setting SPiN Options ............................... 15
   5.3 Model Checking ..................................... 17
   5.4 Error Messages ..................................... 18

6 Verification Experiment ................................ 22

7 Specifying Invariants ................................ 24
   7.1 Assertion in the Code ............................... 24
   7.2 Invariant in the main Method ....................... 24
      7.2.1 Directly as an Assertion ........................ 24
      7.2.2 Indirectly by Calling an Invariant Method .... 25
   7.3 Invariant as a Thread ............................... 26

8 Features of JAVA 1.0 Not Translated ..................... 27
   8.1 Compilation Units .................................. 27
   8.2 New Names .......................................... 27
   8.3 Predefined Types ................................... 27
   8.4 Subclasing .......................................... 27
   8.5 Variable Declarations ................................ 27
      8.5.1 Modifiers ..................................... 27
      8.5.2 Array Declarations ............................ 27
   8.6 Methods and Constructors ........................... 28
      8.6.1 Modifiers ..................................... 28
      8.6.2 Overloading and Overriding .................... 28
      8.6.3 Recursion ..................................... 28
      8.6.4 Names of Called Methods ....................... 28
      8.6.5 Array Argument and Return Types ............... 28
      8.6.6 Actual Parameters in Calls .................... 28
   8.7 Expressions ......................................... 28
      8.7.1 Names .......................................... 28
      8.7.2 Object Instantiations ........................... 29
      8.7.3 Assignment Expressions ........................ 29
      8.7.4 Unary Expressions .............................. 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.7.5 Binary Expressions</td>
<td>29</td>
</tr>
<tr>
<td>8.7.6 Ternary Expressions</td>
<td>29</td>
</tr>
<tr>
<td>8.8 Statements</td>
<td></td>
</tr>
<tr>
<td>8.8.1 Switch Statements</td>
<td>29</td>
</tr>
<tr>
<td>8.8.2 Continue Statements</td>
<td>30</td>
</tr>
<tr>
<td>8.8.3 Labeled Statements</td>
<td>30</td>
</tr>
<tr>
<td>8.8.4 Try-catch-finally Statements</td>
<td>30</td>
</tr>
</tbody>
</table>
1 Introduction

This manual describes Java PathFinder (JPF) [5], a translator from Java [3] [2] [1] to Promela, the programming language of the SPIN model checker [7]. The purpose is to establish a framework for verification and debugging of Java programs based on model checking. The tool is named after the rover operating on Mars in 1997 called the “Mars PathFinder”, and is a play on words: it finds the paths of a Java program that lead to errors.

SPIN is a verification system that supports the design and verification of finite state asynchronous process systems. Programs are formulated in the Promela programming language, which is quite similar to an ordinary programming language, except for certain non-deterministic specification oriented constructs. The SPIN model checker can automatically determine whether a program satisfies a property, and, in case the property does not hold, generate an error trace. SPIN also finds deadlocks.

A significant subset of Java version 1.0 is supported by JPF: dynamic creation of objects with data and methods, class inheritance, threads and synchronization primitives for modeling monitors (synchronized statements, and the wait and notify methods), exceptions, thread interrupts, and most of the standard programming language constructs such as assignment statements, conditional statements and loops. However, the translator is still a prototype and misses some features, such as packages, overloading, method overriding, recursion, strings, floating point numbers, some thread operations like suspend and resume, and some control constructs, such as the continue statement. In addition, arrays are not objects as they are in Java, but are modeled using Promela's own arrays to obtain efficient verification. Finally, we do not translate the predefined class library.

Note that many of these features can be avoided by small modifications to the input code. In addition, the tool is currently being improved to cover more of Java. Despite the omissions, we expect the current version of JPF to be useful on a large class of software. A front-end to the translator checks that the program is in the allowed subset and prints out error messages when not. The translator is developed in Common Lisp, having a Java parser written in MoscowML as front end. A description of an application of JPF to a game server can be found in [6].

The manual is written such that no previous knowledge about the SPIN model checker is needed. All needed SPIN technicalities are described in this document.

The manual is organized around an example buffer program as follows. Section 2 describes how formal specifications of properties are stated as assertions in the Java code. Section 3 describes how one can guide the model checker to do a more efficient job by estimating how many objects are created by each class. Section 4 describes the example program and its specification. Section 5 describes, step by step, the interaction with the system needed in order to model check the Java program. Section 6 gives some examples of errors that can be seeded into the example program, and the expected response by JPF. Section 7 says a bit more about specifications, in particular how class local invariants can be specified. Finally, Section 8 describes the parts of Java 1.0 that currently cannot be translated.
2 Specifications

As will be described in the following, a JAVA program can be annotated with assertions stating boolean properties to be satisfied at certain places in the code. JPF will examine all thread interleavings trying to violate the assertions. In addition to assertion violations, JPF will look for deadlocks. A typical deadlock situation is for example where two threads \( T_1 \) and \( T_2 \) each try to lock two resources \( R_1 \) and \( R_2 \). Suppose \( T_1 \) locks \( R_1 \), and that \( T_2 \) then locks \( R_2 \). If now \( T_1 \) tries to lock \( R_2 \) and \( T_2 \) tries to lock \( R_1 \) a deadlock situation has arisen. JPF will detect situations of this kind independent of whether any assertions have been stated.

In both cases (assertion violation, deadlock) SPiN produces an error trace showing how the corresponding PROMELA program reaches the error state from the initial state. Since there currently is no automatic mapping back to JAVA traces from such PROMELA traces, and since it is very hard to read the PROMELA traces, we have provided a set of print methods with which one can print out essential information from the program execution. This information will be printed on SPIN's graphical message sequence charts as will be illustrated later on.

Assertions and print statements are written in the JAVA code as calls of methods defined in the Verify class shown in figure 1. The Verify class also contains methods for turning JAVA code into atomic blocks executed without interleaving from other threads, and it contains a method for achieving non-determinism. The methods of this class will be described in the following.

Generally, all the methods in the Verify class are defined as static, which means that they can be called by just prefixing the method with the class name as in Verify.assert(....). That is, without instantiating the class into an object. Furthermore, the bodies of these methods are of no importance to the verification, the bodies could be empty, or changed by the programmer to something different. Their contents only have importance when actually running the program on the JAVA Virtual Machine, and hence may have value during normal testing. Note that one is not allowed to add new methods to the Verify class or change the signatures (number of arguments and their types) of the existing methods. The Verify class must be included in the JAVA program in case any of its methods are called. It can be found in the distribution.

2.1 Assertion Methods

The class contains two (overloaded) assert methods, one taking a boolean argument, and one taking a string and a boolean argument. The purpose of these two methods is the same: to check that the argument condition evaluates to true at the place of call. The string only serves a documentary purpose, and will get printed out on the message sequence chart in case the condition is violated. Examples of calls are:

\[
\begin{align*}
\text{Verify.assert(count == 6);} \\
\text{Verify.assert("count != 6",count == 6);} \\
\text{Verify.assert("#7",count == 6);} \\
\end{align*}
\]

2.2 Error Method

In some cases a programmer knows that if control reaches a certain part of the code an error has occurred. At this point one can call the error method, which unconditionally will signal an error, causing the documentary argument text string to be printed out on the message sequence chart. The following two statements are logically equivalent:

\[
\begin{align*}
\text{if (x == 0) Verify.error("x is zero");} \\
\text{Verify.assert("x is zero",x != 0);} \\
\end{align*}
\]
class Verify{
    public static void assert(boolean b){
        if (!b) System.out.println("*** assertion broken");
    }
    public static void assert(String s, boolean b){
        if (!b) System.out.println("*** assertion broken : "+ s);
    }
    public static void error(String s){
        System.out.println("*** error : "+ s);
    }
    public static void print(String s){
        System.out.println(s);
    }
    public static void print(String s, int i){
        System.out.println(s +": "+ i);
    }
    public static void print(String s, boolean b){
        System.out.println(s +": "+ b);
    }
    public static void beginAtomic(){
    }
    public static void endAtomic(){
    }
    public static int random(int max){
        java.util.Random random = new java.util.Random();
        int next = random.nextInt();
        if (next < 0) next = -next;
        return (next % (max + 1));
    }
}

Figure 1: The Verify class

2.3 Print Methods

As mentioned earlier, the Verify class provides a collection of print methods with which one can print out information on SPIN's message sequence charts to illustrate error traces. Each of these methods takes a string argument, which will be printed out, and in addition, two of the methods take a value to be printed, either an integer or a boolean. Examples of calls are:

    Verify.print("method HALT called");
    Verify.print("value of temp", x);
    Verify.print("temperature ok", temp < 80);

Note that a boolean will be printed as a 0 (false) or 1 (true).

2.4 Atomic Execution Methods

In some situations it may be needed to cut down the search space in order to verify a program within reasonable space and time. Code that is surrounded by calls of the methods beginAtomic()
and endAtomic() will be executed in an atomic mode without interleaving from other threads being allowed. The following example illustrates how the initialization of an array is made atomic:

```java
public void initialize(){
    Verify.beginAtomic();
    for (x = 0; x < 10, x++)
        my_array[x] = -1;
    Verify.endAtomic();
}
```

Note that making code execution atomic in this way typically cuts down the state space, but that it may remove the possibility for certain interleaving errors to occur. It is, however, safe to apply around code that effects only local variables not accessible from other threads. But even in the unsafe case the technique may be useful to "hunt down" an error that one has suspicion about, or simply for doing various kinds of experiments on a model that is too big for efficient verification.

The difference between using these two methods and using the synchronized keyword of JAVA is that the latter makes thread execution atomic with respect to a single object. In contrast, the code between beginAtomic() and endAtomic() will be executed atomically with respect to the whole program.

2.5 Non-Deterministic Switch Statements

The random method allows for writing non-deterministic switch statements. It returns a random number between 0 and max (both numbers included). It can only be called in a switch statement as follows (and nowhere else):

```java
switch (Verify.random(2)){
    case 0 :
        x = true;
        break;
    case 1 :
        y = true;
        break;
    case 2 :
        x = true;
        y = true;
        break;
}
```

When executing this program one of the three entries will be non-deterministically chosen (due to the coding of the random method). When translating it using JPF a non-deterministic construct will be generated. For this translation it does not matter what the argument to the random method is - it has absolutely no effect on the translation, and is only used when executing the program. Note that the switch statement is generally supported and will get the normal meaning if the branching expression is not Verify.random(...).
3 Verification Parameters in the Code

A Java program may create an arbitrary number of objects of any of its declared classes. The model checker, however, only allows at most 2 objects of any class to be created, unless otherwise stated in the Parameters class by the user, as described in the following.

Suppose a program contains three classes: C1, C2, and C3. Suppose furthermore that we expect 6 objects to be created of class C1, 1 object to be created of class C2, and at most 2 objects of class C3. We can specify this in the Parameters class as done in Figure 9, where constants C1_size and C2_size specify our estimates of the number of objects created from classes C1 and C2. The size of C2 (1, which is less than the default 2) need not be stated, but stating it will reduce the size of the pre-declared data area for this class.

```java
class Parameters{
    static final int C1_size = 6;
    static final int C2_size = 1;
}
```

Figure 2: The Parameters class
4 A Program Example: The Bounded Buffer

JPF will be illustrated by analyzing a complete, small, but non-trivial Java program. This program, included amongst the examples distributed, is described in the following, together with a specification of its desired properties. The program contains an error, later to be detected by the tool. In order to generate error traces on message sequence charts, Verify.print(...) statements have been inserted in relevant places in the code.

4.1 The Buffer Class

The Java program that we are interested in verifying properties about is a bounded buffer, represented by a single class. An object of this class can store objects of any kind (objects of subclasses of the general top level class Object). Figure 3 shows the declared interface of this class. It contains a put method, a get method and a halt method. Typically there will be one or more producer threads that call the put method, and one or more consumer threads that call the get method. The halt method can be invoked by a producer to inform consumers that it will no longer produce values to the buffer. Consumers are allowed to empty the buffer safely after a halt, but if a consumer calls the get method after the halt method has been called, and the buffer is empty, an exception object of class HaltException will be thrown. A class is an exception class if it is a subclass of the class Throwable. In particular, class Exception is a subclass of Throwable.

```java
class HaltException extends Exception{
}

interface BufferInterface {
    public void put(Object x);
    public Object get() throws HaltException;
    public void halt();
}
```

Figure 3: The Buffer interface

Figure 4 contains the Buffer class annotated with line numbers for later reference. The class declares an array of length 3 to hold the objects in the buffer. In addition to the array, a couple of pointers are declared, one pointing to the next free location, and one pointing to the next object to be returned by the get method. The variable usedSlots keeps track of the current number of elements in the buffer. Finally, the variable halted will become true when the halt method is called.

The three methods of the class are all synchronized (note the synchronized keyword). Hence, each of these methods will have exclusive access to the object when executed. That is, when one of these methods is called on the buffer object by a thread, the buffer gets locked to serve that thread, and it is unlocked again at the end of the method call. The put method takes as parameter the object to be stored in the buffer and has no return value (void). It enters a loop testing whether the buffer is full (i.e. having 3 elements) in which case it calls the built-in wait method. Calling the wait method within a synchronized method suspends the current thread and allows other threads to execute synchronized methods on the object. Such another thread can then call the notify method which will wake up an arbitrarily chosen waiting thread to continue past its wait() call. The notifyAll method wakes up all such waiting threads.

When finally the put method gets past the while loop, it is known that the buffer has free space, and the insertion of the new object can be completed. In case the buffer was in fact empty, all waiting consumers are notified.

The get method is a little bit more complicated because it also takes into account whether the buffer has been halted. Basically, it will wait until there is something in the buffer, and return this element, unless the buffer is empty and at the same time has been halted. In this case, a
1. class Buffer implements BufferInterface {
2.     static final int SIZE = 3;
3.     protected Object[] array = new Object[SIZE];
4.     protected int putPtr = 0;
5.     protected int getPtr = 0;
6.     protected int usedSlots = 0;
7.     protected boolean halted;
8. 
9.     public synchronized void put(Object x) {
10.        while (usedSlots == SIZE) 
11.           try {
12.               Verify.print("producer wait");
13.               wait();
14.           } catch(InterruptedException ex) {};
15.      Verify.print("put", putPtr);
16.      array[putPtr] = x;
17.      putPtr = (putPtr + 1) % SIZE;
18.      if (usedSlots == 0) notifyAll();
19.      usedSlots++;
20.    }
21. 
22.    public synchronized Object get() throws HaltException {
23.        while (usedSlots == 0 & !halted) 
24.           try {
25.               Verify.print("consumer wait");
26.               wait();
27.           } catch(InterruptedException ex) {};
28.        if (halted) {
29.            Verify.print("consumer gets halt exception");
30.            HaltException he = new HaltException();
31.            throw(he);
32.        }
33.    }
34. 
35.    public synchronized void halt() {
36.        Verify.print("producer sets halt flag");
37.        halted = true;
38.        notifyAll();
39.    }
40. }
41. }

Figure 4: The Buffer class
HaltException is thrown. Otherwise, the next buffer element is returned, and producers are notified if the buffer beforehand was full, in which case they may be waiting.

4.2 Setting up an Environment

In order to verify properties about this class, without looking at a complete application within which it occurs, we can create a small application using the buffer. We say that we set up an environment consisting of a number of threads accessing the buffer, and then we prove properties about this small system. This can be regarded as unit testing the buffer. Concretely, we shall define two thread classes: a Producer and a Consumer class, and then start the whole system as shown in the main method in Figure 5.

```java
class Main {
    public static void main(String[] args) {
        Buffer b = new Buffer();
        Producer p = new Producer(b);
        Consumer c = new Consumer(b);
    }
}
```

Figure 5: The main program

First, in order to illustrate the translator's capabilities to translate inheritance, we define the objects that are to be stored in the buffer, see Figure 6. A class Attribute is defined, which contains one integer variable. The constructor method with the same name as the class takes a parameter and stores it in this variable. The class AttrData extends this class with another field, and defines a constructor, which takes two parameters, and then calls the super class constructor with the first parameter.

```java
class Attribute{
    public int attr;

    public Attribute(int attr){
        this.attr = attr;
    }
}

class AttrData extends Attribute{
    public int data;

    public AttrData(int attr,int data){
        super(attr);
        this.data = data;
    }
}
```

Figure 6: The Attribute and AttrData classes

The producer and consumer threads that are actually going to use the buffer are defined in figures 7 and 8. The Producer class extends the Thread class, which means that it must have a run method, which is then executed when an object of this class is started with the start method. As can be seen, the constructor of the class in fact calls this start method in addition to storing locally the buffer for which elements will be produced. The run method adds 6 AttrData objects to the buffer, with attributes 0...5 (in that order) and squares as data, and then calls the halt method on the buffer.
The Consumer class also extends the Thread class. The run method stores all received objects in the received array (or at most 10 of them). Note how the receiving loop is written inside a try construct, which catches and prevents a HaltException from going further.

```java
class Producer extends Thread {
    static final int COUNT = 6;
    private Buffer buffer;

    public Producer(Buffer b) {
        buffer = b;
        this.start();
    }

    public void run() {
        for (int i = 0; i < COUNT; i++) {
            AttrData ad = new AttrData(i, i*i);
            buffer.put(ad);
            yield();
        }
        buffer.halt();
    }
}
```

Figure 7: The Producer class

4.3 Property Specification

Two assertions in Figure 8 state the properties we want to verify. The first assertion states that the consumer receives exactly 6 elements from the buffer. The second assertion within the for loop states that the received elements are the correct ones (at least wrt. the attr value). In addition to these assertions, the system will look for deadlock situations, which need no explicit specification by the user.

4.4 Predicting Number of Objects Created

As can be seen from Figure 7, the producer creates 6 AttrData objects. Since the default number of objects that can be created is 2, we need to specify 6 as the new limit. This is done in the Parameters class in Figure 9.
class Consumer extends Thread {
    private Buffer buffer;

    public Consumer(Buffer b) {
        buffer = b;
        this.start();
    }

    public void run() {
        int count = 0;
        AttrData[] received = new AttrData[10];
        try{
            while (count < 10){
                received[count] = (AttrData)buffer.get();
                count++;
            }
        } catch(HaltException e){};
        Verify.print("Consumer ends");
        Verify.assert("count != COUNT", count == Producer.COUNT);
        for (int i = 0; i < count; i++){
            Verify.assert("wrong value received", received[i].attr == i);
        }
    }
}

Figure 8: The Consumer class

class Parameters{
    static final int AttrData_size = 6;
}

Figure 9: The Parameters class
5 A Guided Tour

This section describes how one model checks a JAVA program. The reader is assumed to know how one generally writes, compiles and executes JAVA programs. In addition, the system must have been installed as described in the installation guide. This means basically that the following two scripts must be available:

- xspin: the graphical interface to SPIN.
- jpf: the JAVA to PROMELA translator.

5.1 Compiling, Executing and Translating the Program

The JAVA program must be stored in a file of the form: myprogram.java. First you should compile the program to check that it is a valid Java program in the first place:

```
javac myprogram.java
```

Compilation should always be applied for the purpose of type checking the program before the translator is applied, since the translator assumes a well-formed JAVA program.

Try now to execute the program by typing:

```
java Main
```

The most likely result will be the following:

```
$ java Main
put : 0
get : 0
consumer wait
put : 1
get : 1
consumer wait
put : 2
get : 2
consumer wait
put : 0
get : 0
consumer wait
put : 1
get : 1
consumer wait
put : 2
get : 2
consumer wait
producer sets halt flag
consumer gets halt exception
Consumer ends
```

We see how the `verify.print` statements in the code print out on the standard output, and the program terminates normally. Note, however, that due to "non-deterministic" scheduling of Solaris threads (into which JAVA threads are mapped when the JAVA Virtual machine is version 1.1.5 or newer), the execution may in fact choose a different path, and break an assertion since our program is bugged. The chance that it does is, however, very low (less than 5% according to our measures). Hence, the chance of catching the error by normal testing is equally low. Try to see if you can break the assertion by executing the program several times.

Now to translate the program into PROMELA, type:
Note that you should not give any .java suffix. This call will translate the file myprogram.java into a PROMELA program, which is written to the file:

myprogram.spin

This file can now be loaded into XSPIN. In the following we describe how to set up XSPIN and how to load and model check the program in myprogram.spin.

5.2 Setting SPIN Options

XSPIN is called as follows:

xspin

As a result, a SPIN window pops up as shown in figure 10 (the body of the window will, however, be initially empty). From now on, the interface is SPIN's interface. To load the translated PROMELA program select FILE:Reopen. This will result in the window shown in the figure.

![Figure 10: SPIN main window](image)

The first time the SPIN window appears a number of options have to be set before model checking can be started. We shall go through these options in the following. If you click on the RUN button, a menu with the following possibilities, amongst others, appear:

- Set Simulation Parameters
- Set Verification Parameters
- (Re)Run Verification

15
First click on RUN: Set-Simulation-Parameters. This brings up the window shown in Figure 11. You should change one on-off field, namely the Data-Values-Panel field, to off as shown on the figure. This is to avoid slow down of the simulator while printing error traces generated by the model checker. Then click on the Cancel button (the option will actually be set as specified).

![Figure 11: SPIN simulation options](image)

Now, in the SPIN window (Figure 10) click on RUN: Set-Verification-Parameters. This brings up the window shown in Figure 12. Turn off the Report-Unreachable-Code option as shown in the figure.

![Figure 12: SPIN verification options](image)

Next, click on Set-Advanced-Options. This will create the window shown in Figure 13. Enter a -J in the Extra-Run-Time-Options field as shown on the figure. Also, activate Use-Compression. Then remove the two verification option windows by hitting respectively Cancel (figure 12), and Set (figure 13). This will set the options.

We shall just mention a few other parameters that may have importance. First of all, the
Supertrace/Bitstate option (Figure 12) can be turned on. This will cause some states to be ignored during model checking, hence reduce time and memory consumption. Of course, some errors may then not be caught. This option can be used when the model is too big for verification.

Second, one of the two options Safety (assertion violations) and Invalid-Endstates (deadlocks) (Figure 12) may be turned off. For example, one may want to look only for deadlocks, ignoring the assertions in the program.

Third, one can experiment with the Physical-Memory-Available, Estimated-State-Space-Size, and Maximum-Search-Depth (Figure 13). Use the explain buttons to get an explanation.

5.3 Model Checking

To activate the SPIN model checker click on RUN:(Re)Run-Verification in the SPIN window (Figure 10). SPIN will now compile the PROMELA program into a C program, which when executed will do the model checking. While the compilation into C takes place, a small window pops up with the text: “Please wait until compilation of the executable produced by spin completes”. When this window disappears, the model checker starts executing (the now generated C program). When this terminates, a window with the verification result appears as shown in Figure 14.

In our case it says (top line): “assertion violated”, and further down it states: “errors : 1”. If there are no errors “errors : 0” is printed. If there is an error, as in our case, an extra window pops up as shown in Figure 15, which suggests actions to be taken at this point. SPIN has created an error trace, leading from the initial program state to the state violating the assertion. We shall now simulate that error trace. Hit Run-Guided-Simulation. That will create yet the window shown in Figure 16. Hitting the Run button in that window will now cause the error trace to be executed (simulated), whereby the Verify.print statements will now cause printing on a graphical message sequence chart, as shown in Figure 17.

This message sequence chart can now be interpreted as follows. The producer (center vertical line) puts 3 values into the buffer, in positions 0, 1 and 2. Then it calls the wait method. The consumer gets the first value (in position 0) and then notifies the producer, indicated by a (red) arrow. Then the consumer gets the two remaining values and waits. The producer then puts the fourth value into position 0 (recall that the buffer is circular), etc. At the end, the producer puts the last sixth value in position 2 and notifies the consumer. Then the producer sets the halted flag. When the consumer now calls the get method, the condition halted in line 27:

```c
if (halted) {
```
will evaluate to true, and an exception will be thrown. Hence, the consumer does not get the last value. The correct code for this line is:

```java
if (usedSlots == 0) {
```

You may want to correct the program, and then try inserting your own errors, or the ones suggested in Section 5.4.

Note that if the JAVA program does not contain any `Verify.print` statements nothing will get printed on the message sequence chart. SPIN will print a PROMELA error trace in the window shown in Figure 16, but this PROMELA trace is hard for a human to relate back to the JAVA program, and is not recommended as a source of information.

### 5.4 Error Messages

Even if program compilation is successful, JPF may still yield error messages. There are two sources of such.
First, the program may be a valid JAVA 1.1 program, but not a valid JAVA 1.0 program. An error message should be printed out if this is the case. The basic difference between JAVA 1.0 and JAVA 1.1 is the new notion of inner classes. JPF cannot handle inner classes.

Second, in case the program is a valid JAVA 1.0 program, the translator checks that it is within the subset being translated. If not, error messages are printed out on the file to which the PROMELA program would normally be written. Hence, just click on the Reopen button in the SPIN window (Figure 10) as usual to load the result of the translation. Any error messages will appear clearly at the beginning of the loaded file. As an example, suppose we write a JAVA program containing the following class definition:

```java
class Operators{
    int x;
    public void shift_left(int i){
        x <<= i;
    }
}
```

using an assignment operator (<<=) not supported by the translator. When translating and then clicking on Reopen in the SPIN window, the contents of that window will appear as shown in Figure 18. For each error it is indicated in which class it occurs, and in case it occurs in a method, also which method.
Figure 17: Message Sequence Chart for assertion violation
Figure 18: Error messages when outside translated subset
6 Verification Experiment

In order to illustrate the effectiveness of JPF (and SPIN of course) we have seeded 21 errors in the program shown in Figure 4, and for each error analyzed the now incorrect program using JPF.

First, however, we have corrected the program as indicated in Section 5.3 by changing line 28 to:

```java
if (usedSlots == 0) {
    ...
}
```

This yields a correct program as starting point. The result of the experiment is shown in Table 1. For each error we give the line numbers changed, referring to Figure 4, and the new contents of these lines. As an example, error 1 is obtained by changing line 4 to “protected int putPtr = 1;” (initializing to 1 instead of to 0). Error 14 was the one seeded into the program from the start.

The results of applying JPF are shown in the fourth column. That is, the result of applying the SPIN model checker to the PROMELA code generated by JPF. The possible outcomes are deadlock (D) and any of the two assertions being violated (A1 referring to the first occurring “count == 6” and A2 referring to the second “received[i].attr == i”). The point here is that all the errors are caught.

The last two columns show the result of running the modified JAVA program on two versions of the JAVA Virtual Machine (JVM) in order to see whether plainly executing the program would highlight the errors seeded. JVM version 1.1.3 is an older version being very deterministic. This means that executing a multi-threaded program several times typically yields the same result every time. JVM 1.1.6 is the newer version with native threads, where JAVA threads are mapped to Solaris threads. This version is therefore non-deterministic, potentially yielding different results for different runs of a multi-threaded program.

Every program has been run several times (from 30 to 100), and the numbers indicate the percentage of runs that have highlighted the error, either via a deadlock, an assertion violation, or a thrown NullPointerException.

All runs, model checking as well as JVM runs, have been executed on a Sun Ultra Sparc 60 with 512 Mb of main memory, and with the Solaris operating system version 5.5.1.

Running the SPIN model checker on the PROMELA code generated by JPF typically used less than half a second to find an error and explored between 40 and 400 states and a similar number of transitions. In a few cases (error 8 and 10) approximately 10000 states and 18000 transitions were explored in less than 2 seconds.

“Errors” 11 and 20 are special (marked with a +) in the sense that they are not really errors when using the environment described in Section 4.2. This environment only creates one consumer, and to make the errors manifest themselves, we needed to create two consumers as shown below:

```java
class Main {
    public static void main(String[] args) {
        Buffer b = new Buffer();
        Producer p = new Producer(b);
        Consumer c1 = new Consumer(b);
        Consumer c2 = new Consumer(b);
    }
}
```

In addition, with two consumers the assertions make no sense and were deleted. Hence, we were not just looking for deadlocks. The table rows for these errors show the result of verifying and executing in this changed multi-consumer environment. The verification of error 11 needed as much as 8 minutes, exploring 2.4 million states and 6 million transitions before the deadlock was found. We verified a down scaled version of this error, with a buffer size of 2 (instead of 3) and the producer only producing 3 values (instead of 6). Also here the deadlock was found by the model checker, but now using 1 minute, and exploring 423096 states and 1 million transitions.
### Table 1: Verification results

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Line</th>
<th>Modification (changed to)</th>
<th>JPF</th>
<th>JVM 1.1.6</th>
<th>J VM 1.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>protected int putPtr = 1;</td>
<td>A2</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>protected int usedSlots = 1;</td>
<td>A1, A2</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>while (usedSlots != SIZE)</td>
<td>D</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>while (usedSlots == 2)</td>
<td>D</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>putPtr = (putPtr + 1) % SIZE;</td>
<td>A2</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>array[putPtr] = x;</td>
<td>A2</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>17</td>
<td>putPtr = putPtr % SIZE;</td>
<td>A2</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>putPtr = (putPtr + 1) % 2;</td>
<td>A2</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
<td>if (usedSlots == SIZE) notifyAll();</td>
<td>D</td>
<td>33</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>18</td>
<td>remove:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>18</td>
<td>if (usedSlots == 0) notifyAll();</td>
<td>D</td>
<td>55</td>
<td>100</td>
</tr>
<tr>
<td>12</td>
<td>23</td>
<td>while (usedSlots == 0)</td>
<td>D</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>13</td>
<td>28</td>
<td>if (usedSlots != 0) {</td>
<td>A1, D</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>28</td>
<td>if (halted) {</td>
<td>A1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>37</td>
<td>if (usedSlots == 0) notifyAll();</td>
<td>D</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>37</td>
<td>remove:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>37</td>
<td>if (usedSlots == SIZE) notifyAll();</td>
<td>D</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>38</td>
<td>usedSlots--;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>38</td>
<td>if (usedSlots == SIZE) notifyAll();</td>
<td>D</td>
<td>66</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>45</td>
<td>notify();</td>
<td>D</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>21</td>
<td>45</td>
<td>remove:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>halted = true;</td>
<td>D</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
7 Specifying Invariants

As described in Section 2 a JAVA program can be annotated with assertions placed in the code in relevant positions. When execution (by the model checker for example) "hits" the assert statement, the condition will be checked. That is, the assertion will only be checked when "it gets its turn to execute".

Suppose we instead want to state a general invariant about part of the variables in the program. That is, suppose we for example want to verify that the value of the variable usedSlots in the Buffer class is always less than or equal to 2 (which is wrong, it can become 3). We shall see solutions of how to specify and verify this in the following.

7.1 Assertion in the Code

The first solution follows a standard strategy of inserting an assertion in the code where the variable usedSlots is incremented, as illustrated in Figure 19. The changed (added) line begins with the symbol "*".

```java
public synchronized void put(Object x) {
    while (usedSlots == SIZE)
        try {
            Verify.print("producer wait");
            wait();
        } catch (InterruptedException ex) {};
    Verify.print("put", putPtr);
    array[putPtr] = x;
    putPtr = (putPtr + 1) % SIZE;
    if (usedSlots == 0) notifyAll();
    usedSlots++;
    Verify.assert("usedSlots > 2", usedSlots <= 2);
}
```

Figure 19: Invariant as assertion in the put method

This solution has the advantage that as soon as the assertion is broken (if broken), the model checker will detect this. Hence, there is a slight advantage wrt. verification time used to locate an error. Also, the generated error trace typically also is shorter than when using the alternative techniques to be described in the following.

The disadvantage of this technique is course that we have to figure out where the variable is updated. In case of mere complicated invariants involving more than one variable, this may become messy and error prone. The remaining solutions do not have this disadvantage, but they may require more time to locate an error.

7.2 Invariant in the main Method

An alternative solution is to place the assertion in the main method after having started all threads. There are two ways of doing this as described in the following.

7.2.1 Directly as an Assertion

Figure 20 illustrates how the assertion is inserted at the end of the main method, after all objects have been created and all threads have been started. The assertion now refers to the variable b.usedSlots, hence, it refers to the variable through the object b.

The way it works is as follows. The main method will start all the threads, and then continue itself as a thread running in parallel with these other threads. In particular, it will be ready to
execute the assert statement at any time. The model checker will therefore execute it in any state. Hence the assertion will function as an invariant that has to hold at any time.

class Main {
    public static void main(String[] args) {
        Buffer b = new Buffer();
        Producer p = new Producer(b);
        Consumer c = new Consumer(b);
        * Verify.assert("usedSlots > 2", b.usedSlots <= 2);
        }
    }

Figure 20: Invariant as assertion in the main method

A disadvantage of this technique is that the state contents of the Buffer class is revealed in the main program: it for example only works if the variable usedSlots is accessible from outside the class. In this case it is since it is protected, and such a variable is visible within the package. Had it, however, been private, we could not have used this technique. The next solution solves this problem, and is just as efficient.

7.2.2 Indirectly by Calling an Invariant Method

In order to make an invariant local to a class, one can state it in a method defined in the class, and then call this method in the main method. This is illustrated in Figure 21. Note that we can name the invariant method as we like, and it can also contain as many assertions as we like. We can even define several invariant methods, and call only some of them at convenience.

class Buffer implements BufferInterface {
    static final int SIZE = 3;
    protected Object[] array = new Object[SIZE];
    protected int putPtr = 0;
    protected int getPtr = 0;
    protected int usedSlots = 0;
    protected boolean halted;
    *
    public void invariant() {
        * Verify.assert("usedSlots > 2", usedSlots <= 2);
        *
    }

    ...

    ...

class Main {
    public static void main(String[] args) {
        Buffer b = new Buffer();
        Producer p = new Producer(b);
        Consumer c = new Consumer(b);
        b.invariant();
        }
    }

Figure 21: Invariant as method call in the main method

The advantage of this method is that now the invariant is really local to the class. The disad-
vantage is that one still needs to call the invariant method in the main program, which again means that the buffer object b must be visible in the main program. The last solution solves this problem.

7.3 Invariant as a Thread

The final solution consists of defining the Buffer class as an extension of the Thread class, and then define the required run method to contain a call to the invariant method. This is shown in Figure 22. The figure shows how the Buffer class has been extended with an invariant method, a run method that calls this invariant, and a constructor that starts the thread. Hence, whenever a Buffer object is created with the new method, a thread is started that at any time may check the assertion. Again, several invariant methods can be defined and called. One can also write the assertions directly in the run method.

```java
class Buffer extends Thread implements BufferInterface {
    protected Object[] array = new Object[SIZE];
    protected int putPtr = 0;
    protected int getPtr = 0;
    protected int usedSlots = 0;
    protected boolean halted;

    public void invariant() {
        Verify.assert("usedSlots > 2", usedSlots <= 2);
    }

    public void run() { invariant(); }

    public Buffer() { this.start(); }
}
```

Figure 22: Invariant as a thread

The disadvantage of this technique is that it only works for passive classes that are not already extensions of the Thread class.
8 Features of JAVA 1.0 Not Translated

JPF translates a subset of JAVA 1.0, and this section identifies the features not translated, each devoted a subsection in the following. JAVA 1.0, in turn, is a subset of JAVA 1.1, which in addition provides, amongst other things, inner classes. Consequently, JPF does for example not translate inner classes. A good and relatively short description of JAVA 1.0, and the extension JAVA 1.1, can be found in [2].

The translator should print out error messages if the JAVA program is not within the translated subset. In the following cases, however, no error messages will be printed even though these features are not translated: method overloading, method overriding, and method recursion. The SPIN syntax checker will, however, complain and reject the translated PROMELA program. Since SPIN's error messages may not be easily related to the JAVA program it should preferably be reassured by human inspection that the JAVA program does not use any of these features.

8.1 Compilation Units

A JAVA program must consist of a single package. References to other packages, user-defined as well as predefined JAVA packages, such as java.lang, are not allowed. Consequently, import declarations are not allowed.

As a consequence, class names must either refer to user-defined classes or to one of the predefined classes: Object, Thread, or Exception.

8.2 New Names

A user-defined name cannot end with underscore: '_'. The reason for this is that internal names generated by the translator end with '_'.

8.3 Predefined Types

The following primitive types are not allowed: char, float, and double. The type String is furthermore not allowed. Literals of these types are not allowed to occur in the program.

8.4 Subclassing

In a class declaration of the form:

    class SomeClass extends SuperClass { ... }

the SuperClass must either be a user-defined class or one of the pre-defined JAVA classes: Thread or Exception.

8.5 Variable Declarations

8.5.1 Modifiers

The native modifier is not allowed in instance variable declarations.

8.5.2 Array Declarations

Arrays must be one-dimensional and must be given a dimension at declaration time in terms of an integer literal, or alternatively in terms of an array initializer, as in the following examples:

    static final int SIZE = 7;

    int[] ia = new int[5];
    Object[] oa = new Object[SIZE];
    byte[] ba = {1,9,9,9};
Note that if a constant (static and final) is used to define the size of an array, this constant cannot be defined in terms of other constants - it has to be defined directly representing some integer literal as in the above case (note that this is only a restriction on constants used in defining array dimensions). Hence, the following is illegal:

illegal:

\[
\text{static final int BIGGER\_SIZE} = \text{SIZE + 1;}
\]

\[
\text{int[]} b = \text{new int[BIGGER\_SIZE];}
\]

The array brackets [] have to occur in combination with the type. For example, the following declaration where the [] brackets are associated with the array variable a is not allowed:

illegal:

\[
\text{int a[]} = \text{new[5];}
\]

8.6 Methods and Constructors

8.6.1 Modifiers

The following method modifiers are not allowed: abstract and native.

8.6.2 Overloading and Overriding

Methods and constructors cannot be overloaded and methods cannot be overridden. As a consequence there can be at most one constructor per class. However, each subclass of a class can have its own constructor.

8.6.3 Recursion

Recursive methods are not allowed.

8.6.4 Names of Called Methods

Except for user-defined methods only the following pre-defined JAVA methods can be called:

\[
\text{start stop sleep yield wait notify notifyAll}
\]

8.6.5 Array Argument and Return Types

The type of a method or constructor argument cannot be an array type. Similarly, the result type of a value returning method cannot be an array type.

8.6.6 Actual Parameters in Calls

Actual parameters in method calls cannot themselves be method calls.

8.7 Expressions

8.7.1 Names

Names occurring in expressions or on left-hand sides of assignment statements must be user-defined.
8.7.2 Object Instantiations
An object instantiation using the new operator is only allowed in association with a variable
declaration or in an assignment statement, as illustrated here:

```java
SomeClass s = new SomeClass(42); // in a declaration
OtherClass t;
t = new OtherClass(43); // in an assignment statement
```

Similarly, an array instantiation using the new operator can only (and must) occur in an array
declaration as discussed in Section 8.5.2.

8.7.3 Assignment Expressions
Assignments are not allowed as expressions. For example, the following is not allowed:

```java
illegal:
x = (y = y + 1);
```

8.7.4 Unary Expressions
The following unary expressions are not allowed:

```java
illegal:
--expr ++expr expr-- expr++ ~expr
```

The first four of these are, however, allowed as statements!

8.7.5 Binary Expressions
The following binary operators are not allowed:

```java
illegal:
<< >> >>> instanceof
```

8.7.6 Ternary Expressions
Ternary expressions of the form: \( (expr_1?expr_2:expr_3) \) are not allowed.

8.8 Statements

8.8.1 Switch Statements
Each entry statement not being the last in a switch statement should exit with a break or return
as for example in the following program:

```java
switch (x){
    case -1 :
    case 0 :
    case 1 : near_zero = true; break;
    default : near_zero = false
}
```
This statement will assign true to near_zero if \( x \) is either \(-1, 0\) or \(1\), and otherwise false. Now, if we remove the break statement, and \( x \) has the value \(-1, 0\) or \(1\), it will first assign true and then false, resulting in false. The switch statement is said to "fall through" from the first case to the second case. This behavior is not supported, and the translator will insert the break statements if they are not present.

8.8.2 Continue Statements

Continue statements are not allowed.

8.8.3 Labeled Statements

Labeled statements are not allowed.

8.8.4 Try-catch-finally Statements

A catch in a try statement is only allowed to catch exceptions of user-defined exception classes or exceptions of the classes Exception, InterruptedException or ThreadDeath, the latter being thrown by the stop method.
References


