Int J STTT (2000) 2: 366–381

# Model checking JAVA programs using JAVA PathFinder

**Klaus Havelund, Thomas Pressburger**

NASA Ames Research Center, Recom Technologies, Moffett Field, CA, USA; E-mail: {havelund,ttp}@ptolemy.arc.nasa.gov

**Abstract.** This paper describes a translator called JAVA PATHFINDER (JPF), which translates from JAVA to PROMELA, the modeling language of the SPIN model checker. JPF translates a given JAVA program into a PROMELA model, which then can be model checked using SPIN. The JAVA program may contain assertions, which are translated into similar assertions in the PROMELA model. The SPIN model checker will then look for deadlocks and violations of any stated assertions. JPF generates a PROMELA model with the same state space characteristics as the JAVA program. Hence, the JAVA program must have a finite and tractable state space. This work should be seen in a broader attempt to make formal methods applicable within NASA's areas such as space, aviation, and robotics. The work is a continuation of an effort to formally analyze, using SPIN, a multi-threaded operating system for the Deep-Space 1 space craft, and of previous work in applying existing model checkers and theorem provers to real applications.

**Key words:** Program verification – JAVA – Model checking – SPIN – Concurrent programming – Assertions – Deadlocks

## 1 Introduction

In this paper we describe JAVA PATHFINDER (JPF), a translator from a subset of the JAVA programming language to PROMELA, the modeling language of the SPIN model checker. JAVA [1, 6] is a general purpose object-oriented programming language with built in mechanisms for multi-threaded programming. SPIN [13] is a verification system that supports the design and verification of finite state asynchronous process systems. Models are formulated in the PROMELA language, which is quite similar to an ordinary programming language, except for certain non-deterministic specification oriented constructs. Processes communicate either via shared variables or via message passing through buffered channels. Properties to be verified are stated as assertions in the code, or as formulae in the linear temporal logic LTL. 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.

JPF translates a given JAVA program into a PROMELA model, which then can be model checked using SPIN. The JAVA program may contain assertions, which are translated to similar assertions in the PROMELA model. The SPIN model checker will then look for deadlocks and violations of any stated assertions. JPF does not apply any analysis to reduce the size of the generated model. Hence, the JAVA program must have a finite and tractable state space. JPF has been applied to programs having up to 2000 lines of code.

The work should be seen in a broader attempt to make formal methods applicable "in the loop" of programming within NASA's areas such as space, aviation, and robotics. Our main long term goal is to create an automated formal methods workbench for JAVA programming so that programmers themselves can apply them in their daily work (in the loop) without the need for specialists to manually reformulate a program in a different notation in order to analyze it. The tool we are developing is named after the rover operating on Mars in 1997 called the "Mars PathFinder". Although this mission was generally regarded as a big success, the rover did in fact contain a number of software bugs (causing repeated rebootings and panic at NASA headquarters) that could potentially have been found beforehand using proper verification tools. The JAVA PATHFINDER name is a play on words: it finds the *paths* of a JAVA program that lead to errors.

In an earlier effort we formally verified, using SPIN, a multi-threaded operating system for a spacecraft [9]. The operating system is one component of NASA's New Millennium Remote Agent (RA) [14], an artificial intelligence based spacecraft control system architecture launched October 24, 1998, as part of the Deep–Space 1 mission to an asteroid to validate the potential of artificial intelligence, ion propulsion, and other technologies for future space crafts. The operating system is implemented in a multi-threaded version of COMMON LISP. The verification effort consisted of hand translating parts of the LISP code into the PROMELA language of SPIN. A total of 5 errors were identified, a very successful result. In addition, one of these errors appeared in a different part of the system not examined using SPIN, and caused a deadlock during flight in space, resulting in thrusting not being turned off when requested to. This was regarded as a clear demonstration of the potential of a system like SPIN.

It is often stated that model checkers of today cannot handle real sized programs, and consequently cannot handle real sized JAVA programs. This is certainly true. However, there are three aspects that make our effort worthwhile anyway. First, by providing an *abstraction workbench* we will make it possible to cut down the state space of a JAVA program. Second, one can imagine model checking being applied for unit testing, where one focuses on a single class (or a few classes) and puts this (these) in parallel with an aggressive environment represented by a number of threads. Finally, JAVA can be used as a design language, just as can PROMELA.

Few attempts have been made to automatically verify programs written in real programming languages. The most recent attempt we can mention is the one reported in [4], which also translates JAVA programs into PROMELA, however not handling exceptions or polymorphism as we do. The work in [2] defines a translator from a concurrent extension of a limited subset of C++ to PROMELA. A disadvantage of this solution is that the concurrency extensions are not broadly used by C++ programmers. *Extended static checking*, is described in [5], is a technique for detecting, at compilation time, common programming errors. The technique uses program verification technology, but feels like a type checker to a programmer. By using an underlying automatic theorem prover, the technique is more semantically thorough than decidable static analysis techniques. At the same time, by only trying to detect certain kinds of errors, not prove the program's correctness, the technique is more automatic than program verification. Finally, [3] describes a theory of translating JAVA programs into transition models, making use of static pointer analysis to aid *virtual coarsening*, which reduces the size of the models. That is, a sequence of transitions in an obtained model can be collapsed into an atomic transition if they only access variables that are either local to a thread or protected by a lock owned by the thread. This work can be seen as complementing our work in the sense that JPF produces a transition model (in PROMELA) while [3] describes how to reduce the size of such a model.

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 (since method calls are translated by inlining the method bodies), 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. 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. As a general observation, JPF translates a bigger subset of JAVA than any similar existing tool that we know of.

The translator is developed in COMMON LISP, having a JAVA parser written in MOSCOWML as front end. Gerard Holzmann supported our efforts by changing the semantics for the PROMELA `unless` construct in order to simplify the translation of exceptions.

The paper is organized around an example JAVA program that has been translated automatically by JAVA PATHFINDER and verified automatically by SPIN. In Sect. 2 we describe this program, a bounded buffer. In Sect. 3 we describe what the resulting PROMELA code looks like. In Sect. 4 we present an experiment where we seeded 21 errors into the example program, and ran the SPIN model checker on the code generated by JPF. Section 5 ends with conclusions and suggestions for future work.

## 2 The bounded buffer program

The translation scheme will be illustrated by translation of a complete, small, but non-trivial JAVA program that covers many of the features of JAVA that we can translate. After translation by JPF, SPIN can be applied to prove or disprove that the program satisfies given properties stated as assertions in the program, and that it is deadlock free.

### 2.1 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 1 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`.

Figure 2 contains the `Buffer` class annotated with line numbers for later reference. A JAVA class is generally speaking described by a name, a set of data variables, zero or more constructor methods (with different argument types if more than one) with the same name as the class, and a collection of methods. One of the constructors is executed when an object is created with the `new` construct. Note that the `Buffer` class has no such user-defined constructors. 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.

```
class HaltException extends Exception{}

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

**Fig. 1.** The JAVA `Buffer` interface

```
 0. class Buffer implements BufferInterface {
 1.   protected static final int SIZE = 3;
 2.   protected Object[] array = new Object[SIZE];
 3.   protected int putPtr = 0;
 4.   protected int getPtr = 0;
 5.   protected int usedSlots = 0;
 6.   protected boolean halted;
 7.
 8.   public synchronized void put(Object x) {
 9.     while (usedSlots == SIZE)
10.       try {wait();}
11.       catch(InterruptedException ex) {};
12.     array[putPtr] = x;
13.     putPtr = (putPtr + 1) % SIZE;
14.     if (usedSlots == 0) notifyAll();
15.     usedSlots++;
16.   }
17.
18.   public synchronized Object get()
19.     throws HaltException{
20.     while (usedSlots == 0 & !halted)
21.       try {wait();}
22.       catch(InterruptedException ex) {};
23.     if (usedSlots == 0) {
24.       throw(new HaltException());
25.     };
26.     Object x = array[getPtr];
27.     array[getPtr] = null;
28.     getPtr = (getPtr + 1) % SIZE;
29.     if (usedSlots == SIZE) notifyAll();
30.     usedSlots--;
31.     return x;
32.   }
33.
34.   public synchronized void halt(){
35.     halted = true;
36.     notifyAll();
37.   }
38. }
```

**Fig. 2.** The JAVA `Buffer` class

The call of `wait` is put inside a `try` construct, which is JAVA's exception handling construct. A general `try` construct has the form:

```
try T
catch(E1 e1) C1
...
catch(E2 en) Cn
finally F
```

where each `T,C1,...,Cn,F` is a block (a statement or a sequence of statements enclosed by {...}) and each `Ei` is an exception type (class). The body `T` of the `try` statement is executed until either an exception is thrown or it finishes successfully. If an exception is thrown, the `catch` clauses are examined from top to bottom in order to find one where the thrown exception is of the corresponding class `Ei` or of a subclass thereof. If such a `catch` is found, the corresponding block `Ci` is executed. If no appropriate `catch` is found, the exception "flows out" of the `try` statement into an outer `try` that might handle it. There can be any number of `catch` clauses in a `try` including none. If no `catch` clause in the method catches the exception, the exception is thrown to the code that invoked this method. If a `finally` clause is present in a `try`, its code is executed after all other processing in the `try` is complete. This happens no matter how the completion was achieved, whether normally, through an exception, or through a control flow statement like `return`.

Normally an exception is thrown explicitly within a thread using the `throw(e)` statement, where `e` is an exception object (a normal object of an exception class which may include data and methods). However, one thread `S` may throw an exception in another thread `T` by executing `T.interrupt()`, which throws an `InterruptedException`, or `T.stop()`, which throws a `ThreadDeath` exception. The `try` construct around the `wait` call is supposed to catch interrupts from other threads. As we see, nothing is done in this case (lines 11 and 22), but the `try` statement is necessary in order for the JAVA type checker to accept the program. We shall later see a real use of exceptions. One of the main results in this paper is that we can in fact prove properties of programs that throw exceptions.

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 `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.

### *2.2 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 Fig. 3.

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 Fig. 4. 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.

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

**Fig. 3.** The JAVA `main` program

```
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;
  }
}
```

**Fig. 4.** The JAVA `Attribute` and `AttrData` classes

The producer and consumer threads that are actually going to use the buffer are defined in Fig. 5. 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 $0 \ldots 5$ (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.

### *2.3 Property specifications*

JPF allows a programmer to annotate his/her JAVA program with assertions and verify them using the SPIN model checker. In addition, deadlocks can be identified.

```
class Producer extends Thread {
  private Buffer buffer;

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

  public void run() {
    for (int i = 0; i < 6; i++) {
      AttrData ad = new AttrData(i,i*i);
      buffer.put(ad);
      yield();
    };
    buffer.halt();
  }
}


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.assert(count == 6);
    for (int i = 0; i < count; i++){
      Verify.assert(received[i].attr == i);}
  }
}
```

**Fig. 5.** The Java `Producer` and `Consumer` classes

An assert statement is expressed as a call to the static method `assert` in the `Verify` class shown in Fig. 6. The fact that it is static means that we can call this method directly using the class name `Verify` as prefix, without making an object instance first. The body of this method is of no real importance for the verification since only the call of this method (and not its definition) will be translated into a corresponding Promela assert statement. A meaningful body, like printing an error message as in this example, can be useful during normal testing though, but it will not be translated into Promela.

The first assertion in Fig. 5 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 with respect to the `attr` value).

```
class Verify{
  public static void assert(boolean b){
    if (!b)
      System.out.println("assertion broken");
  }
}
```

**Fig. 6.** The Java `Verify` class

One can consider other kinds of `Verify` methods, in general methods corresponding to the operators in Ltl, the linear temporal logic of Spin. Since these methods can be called wherever statements can occur, this kind of logic represents what could be called an *embedded temporal logic*. As an example, one could consider a statement of the form: `Verify.eventually(count == 6)` inserted as the second statement of the consumer `run` method. The major advantage of this approach is that we do not need to change the Java language, or parse special "specification comments". Java itself is used as the specification language. Note, however, that at this point, only the `assert` method is supported.

## 3 Translation

### *3.1 Classes, inheritance and object creation*

The general principle behind the translation is the following. A Java class basically consists of *data variables* and *methods*. For each new creation of an object, a new set of data variables, a *data area*, is allocated, and the methods of that object will then work on this newly allocated area. Hence, at any point in time a set of data areas will have been allocated, one for each object not garbage collected. We shall model the set of data areas of a class by an array of records (typedef's in Spin terminology), one record for each data area. An index variable will point to the next free record in the array, initially having the value 0 (first record in the array). Method definitions are mapped into macro definitions, and method calls are mapped into applications of macros. Note that the translated code shown is the unedited Promela code generated by the translator (except for a couple of omissions).

For illustration, we start with the translation of the simplest classes, namely `Attribute` and `AttrData`, which have no methods, but which show inheritance. Figure 7 shows the data areas for the two classes. For each class `C` an array named `C_Obj` is defined of size `MAX_OBJECT` and of type `C_Class`. In addition, a pointer `C_Index` to the next free data area (object) is declared. Because the `AttrData` class inherits from the `Attribute` class, it includes the `attr` variable in addition to the `data` variable.

Note that this translation puts a limit (`MAX_OBJECT`) on the number of objects that can be created of a class.

```
#define Index byte
#define MAX_OBJECT 7

typedef Attribute_Class{
  int attr;
};
Attribute_Class Attribute_Obj[MAX_OBJECT];
Index Attribute_Next = 0;

typedef AttrData_Class{
  int attr;
  int data;
};
AttrData_Class AttrData_Obj[MAX_OBJECT];
Index AttrData_Next = 0;
```

**Fig. 7.** Object arrays for the `Attribute` and `AttrData` classes

A particular object is now referenced by an object reference containing information about which class it concerns (and thereby which array) and which instance (index in the array) it concerns. Because PROMELA's record concept (typedefs) is not flexible enough for our needs (e.g., field access cannot be applied to a conditional expression) we have decided to represent an object reference $(c, i)$ as the integer $c * 100 + i$. The operations (macros) for dealing with these calculations are shown in Fig. 8. Each class is given a unique identification which is a number, for example class `Attribute` is given number `6`.

In order to access the variables in these arrays, macros are defined as shown in Fig. 9. These macros are prefixed with the name of the class that defines the variable, which is statically decidable in JAVA at the point where the variable is accessed (in contrast to method calls). However, because JAVA supports polymorphism where an object may belong to a subclass of the statically declared class (for example, if it is a parameter to a method) we need to define these variable accessors so that they access the right array. For example, the macro `Attribute_get_attr` for reading the `attr` variable defined in the `Attribute` class is defined as a conditional expression over the subclasses.

Both classes have constructors which will be executed when creating new objects. For example, the JAVA expression "`new AttrData(2,4)`" will cause a call of the constructor, which in turn will call the constructor of the super class: "`Attribute(2)`". Constructors are modeled as macros as shown in Fig. 10 (slightly simplified for presentation purposes; the real translation also takes into account variable initializations occurring together with variable declarations). The parameter `obj` will be an object reference denoting the object being constructed.

The construction of a new object such as, for example, the JAVA statement "`x = new AttrData(2,4)`" is mapped into the sequence of PROMELA statements shown in Fig. 11 (slightly simplified here with respect to the treatment of the variable `x`): one can see how

```
...
#define Attribute 6
#define AttrData  7
...

#define create_object(c,i)
  (c*100 + i)

#define get_class(x)
  (x/100)

#define get_index(x)
  (x - ((x/100)*100))
```

**Fig. 8.** General object reference operations

```
#define undefined 0

#define Attribute_get_attr(obj)
  (get_class(obj) == AttrData ->
    AttrData_Obj[get_index(obj)].attr :
  (get_class(obj) == Attribute ->
    Attribute_Obj[get_index(obj)].attr :
  undefined))

#define Attribute_set_attr(obj,value)
  if
  :: get_class(obj) == AttrData ->
       AttrData_Obj[get_index(obj)].attr = value
  :: get_class(obj) == Attribute ->
       Attribute_Obj[get_index(obj)].attr = value
  fi

#define AttrData_get_data(obj)
  AttrData_Obj[get_index(obj)].data

#define AttrData_set_data(obj,value)
  AttrData_Obj[get_index(obj)].data = value
```

**Fig. 9.** Variable accessors for the `Attribute` and `AttrData` classes

```
#define Attribute_constr(obj,attr)
  Attribute_set_attr(obj,attr);

#define AttrData_constr(obj,attr,data)
  Attribute_constr(obj,attr);
  AttrData_set_data(obj,data);
```

**Fig. 10.** Constructors for the `Attribute` and `AttrData` classes

```
atomic{
  x  = create_object(AttrData,AttrData_Next);
  AttrData_Next++
};
AttrData_constr(x,2,4);
```

**Fig. 11.** Translation of `x = new AttrData(2,4)`

the x is bound to a reference, the next-pointer is incremented, and the constructor is applied to the reference now stored in x.

## *3.2 Synchronization on objects*

The Buffer class is more complicated because it contains methods, and also because these are synchronized, which means that they must have exclusive access to the object when executing. That is, only one thread at a time may execute any synchronized method on the same object. Therefore, the translation must provide a locking mechanism with which one can lock an object to serve a particular thread that calls a synchronized method on the object. For that purpose some extra variables are inserted into the data area of each object, as shown in Fig. 12. In addition to the variables declared in the class (note that JAVA arrays are modeled directly as PROMELA arrays), it contains the three variables LOCK, WAITING and WAIT used for managing object locking and release. The LOCK variable will at any time either be null (a negative integer) or it will be the thread id of the thread that currently is executing a synchronized method on the object (actually a non-negative PROMELA process id). Hence, once this field is set to a proper thread id by a thread that calls a synchronized method, only the thread with this thread id is allowed to operate on the object. When the call of the synchronized method terminates, the lock is released by setting it to null again. Note that a synchronized method must be allowed to call other synchronized methods on the same object without causing blocking.

The variables WAITING and WAIT are used to manage threads that call the wait, notify, and notifyAll methods on the object. A thread that calls wait() will first unlock the object, and then try to read a value on the zero place channel WAIT. A zero place channel in PROMELA is used to model rendezvous communication, hence some other thread must send a value on this channel in order for the calling thread to be released. At any time, all threads that are waiting to get access to the object are waiting on this channel. Each time a thread calls the wait() method on the object, the WAITING variable is additionally incremented, and conversely decremented when released. Hence, the value of this variable will always be the number of threads waiting on the object. The variable is used when notifyAll is called by a thread, and all waiting threads have to be released: we need to know how many times the WAIT channel must be signaled. The operations on the WAITING variable are shown in Fig. 13.

Locking and unlocking an object is implemented by the operations shown in Fig. 14. Locking an object is done when a thread calls a synchronized method on an object which the thread has not already locked. First the thread waits until the lock becomes free (no other thread has locked it), which is the case when the LOCK field becomes null. Note that in PROMELA, an equality is a statement that is executable only when the equality holds, and hence blocks until this is the case. Then, in an atomic move, the thread locks the object by assigning the value of "this" to the LOCK field, where "this" denotes the thread id of the current thread ("this" is defined as _pid, which in PROMELA refers to the process id of the currently executing process). Unlocking an object just corresponds to setting the LOCK field to null.

Finally, the macros modeling the wait, notify and notifyAll methods are shown in Fig. 15. The Buffer_wait macro models the wait method. Ignore for a moment the unless constructs. When called, it first unlocks the object, because other threads should now have access to perhaps later call notify. Then the WAITING field is increased, and finally the thread starts waiting for a signal (the value 0) on the WAIT channel. When the thread is awakened by a notify or notifyAll, it locks the object again (first occurring call of Buffer_lock(obj)) in order to continue with exclusive

```
typedef Buffer_Class{
  int LOCK;
  byte WAITING;
  chan WAIT = [0] of {bit};
  ObjRef array[3];
  int putPtr;
  int getPtr;
  int usedSlots;
  bit halted;
};
```

**Fig. 12.** Object type for the Buffer class

```
#define Buffer_any_WAITING(obj)
  Buffer_Obj[get_index(obj)].WAITING > 0

#define Buffer_incr_WAITING(obj)
  Buffer_Obj[get_index(obj)].WAITING++

#define Buffer_decr_WAITING(obj)
  Buffer_Obj[get_index(obj)].WAITING--
```

**Fig. 13.** Operations on WAITING

```
#define this _pid

#define Buffer_lock(obj)
  atomic{
    Buffer_get_LOCK(obj) == null ->
      Buffer_set_LOCK(obj,this)}

#define Buffer_unlock(obj)
  Buffer_set_LOCK(obj,null)
```

**Fig. 14.** Operations for locking and unlocking objects

```
#define continue 0

#define Buffer_wait(obj)
  atomic{
    {
      Buffer_unlock(obj);
      Buffer_incr_WAITING(obj);
      Buffer_get_WAIT(obj)?continue
    }
    unless{
      d_step{
        get_this_EXN > 0 ->
        Buffer_decr_WAITING(obj)
      }
    };
    Buffer_lock(obj)
  }
  unless {a_finally(Buffer_lock(obj))}

#define Buffer_notify(obj)
  atomic{
    if
    :: Buffer_any_WAITING(obj) ->
         Buffer_get_WAIT(obj)!continue;
         Buffer_decr_WAITING(obj)
    :: else -> skip
    fi}

#define Buffer_notifyAll(obj)
  atomic{
    do
    :: Buffer_any_WAITING(obj) ->
         Buffer_get_WAIT(obj)!continue;
         Buffer_decr_WAITING(obj)
    :: else -> break
    od}
```

**Fig. 15.** The operations `wait`, `notify` and `notifyAll`

access to the object (note that `wait` can only be called within a synchronized context in JAVA).

The `Buffer_notify` macro signals the `WAIT` channel in case there are threads waiting for a signal. The `Buffer_notifyAll` macro does this as long as there are threads waiting. Note that this happens atomically so that no new threads can join the waiting threads during the notification.

Now we explain the `unless` constructs in the `Buffer_wait` macro. This extra complication is caused by the fact that another thread can either interrupt or stop a thread that is waiting. Recall that such an interrupt or stop is like an exception being thrown inside the (perhaps waiting) thread being interrupted or stopped. We shall later explain exceptions in more detail, but here it suffices to say that in case such an exception is thrown, the waiting thread must be released immediately; this occurs because the conditions of the two `unless` constructs become true. The PROMELA construct "$P$ `unless` $Q$" executes $P$ to its end unless $Q$ becomes executable before the termination of $P$, in which case $P$ is aborted, and execution continues with $Q$.

Suppose that the waiting thread is blocked in the "`Buffer_get_WAIT(obj)?continue`" statement, and that such an interrupt occurs. Then (due to a new semantics of the PROMELA `unless` construct, see Sect. 3.5) the inner `unless` construct will be triggered since, as we shall see, this implies that `get_this_EXN` becomes bigger than zero. This causes the `WAITING` variable to be decremented, which is necessary because the `Buffer_notify` and `Buffer_notifyAll` macros normally do this (and now have not had the chance to do it). Next, because the waiting thread may have code in `finally` constructs that must be executed before termination (not shown here), it is necessary to lock the object again before stopping, such that this finalize code can be executed with exclusive access to the object. This is done by the outermost `unless` construct. It was necessary to introduce two `unless` constructs since the interrupt may also arrive after the thread has been notified, but while it is waiting for the lock to be released; that is, it is waiting in the first occurring `Buffer_lock(obj)` statement.

## *3.3 Methods*

In order to illustrate how methods are modeled, we illustrate the simplest one, namely the `halt` method. A synchronized method `M` in a class `C` is defined in two parts: a `C_M` macro dealing with object locking, and a `C_M_code` macro containing the actual code of the method. For non-synchronized methods there is no such distinction. This is shown for the synchronized `halt` method in Fig. 16. In general, a method call is translated into a call of the `C_M` macro with parameters corresponding to the method parameters. In the case of `halt`, the method is parameterless, hence only the object reference of the object upon which the method is called is given as parameter. The macro tests whether the object is already locked by the calling thread, in which case the actual code, `C_M_code`, is executed. This models the situation where this method

```
#define Buffer_halt(obj)
  if
  :: Buffer_get_LOCK(obj) == this ->
       Buffer_halt_code(obj)
  :: else ->
       Buffer_lock(obj);
       try(Buffer_halt_code(obj))
         unless {d_finally(Buffer_unlock(obj))}
  fi

#define Buffer_halt_code(obj)
   Buffer_set_halted(obj,true);
   Buffer_notifyAll(obj)
```

**Fig. 16.** Translation of the `halt` method

is called by another synchronized method, hence the object is already locked. In case the object is not locked by the calling thread, an attempt is made to lock it (`Buffer_lock(obj)`), and after successful locking, the code is executed. The `try` macro models the `try` construct of JAVA and models the fact that the object must always be unlocked (`Buffer_unlock(obj)`) when leaving the method, also in the case where an exception is raised during execution of the code.

### *3.4 Threads*

It remains to explain how things are put together by translating the main method and the threads that it starts. First, we describe how the thread classes `Producer` and `Consumer` are translated. We focus on the `Producer`. Thread classes are translated in a similar manner to other classes, with the addition that the `run` method is translated into a PROMELA process type (`proctype`), that can then be started using the PROMELA `run` statement, whenever a corresponding JAVA thread is started with the JAVA `start` method. Figure 17 shows the translation of the `run` method of the `Producer` class of Fig. 5.

The process takes as a parameter its own object reference. Notice how local variables in *called* methods are declared as global variables in the process. For example, the variable `ex` in the `put` method that binds the caught exception in the `try` statement (Fig. 2) is declared at this point. This is necessary because PROMELA does not support local variables (nor does it in the new inline procedures that PROMELA got recently). Of course, this solution prevents the translation of recursive methods, but it is efficient with respect to verification time since we don't need to maintain a call stack (and represents what one normally would do in a hand translation). Another possibility is to translate methods into `proctype`s, as suggested in [13], but experiences during earlier work suggested that this would be inefficient, see [9]. The while loop is translated into PROMELA's `do ... od` construct in a very straightforward way.

Note in Fig. 17 how the object reference to the buffer in the `buffer` variable declared inside the `Producer` class is accessed and passed as argument to the `put` and `halt` methods. This models the dot-notation in JAVA for accessing methods in an object. Surrounding the producer thread code is an `unless` construct, which is supposed to catch any exceptions thrown and not caught by the user program. This will be explained in the next section. Finally, Fig. 18 shows the translation of the `Producer` object constructor. It shows how the `buffer` variable is initialized with the argument, and how the process is started, corresponding to the "`this.start()`" call.

### *3.5 Exceptions*

JAVA exceptions are complicated when considering all the situations that may arise, such as method returns in the middle of `try` constructs, the `finally` construct, interrupts (which are exceptions thrown from one thread to another) of threads that have called the `wait` method, and the fact that objects have to be unlocked when an exception is thrown out of a synchronized method. We shall try to illustrate our solution.

```
proctype Producer_Thread(ObjRef obj){
  {
    int Producer_run_i;
    ObjRef Producer_run_ad;
    ObjRef Buffer_put_ex;
    Producer_run_i = 0;
    do
    :: (Producer_run_i < 6) ->
       atomic{
         Producer_run_ad =
           create_object(AttrData,AttrData_Next);
         AttrData_Next++};
       AttrData_constr(Producer_run_ad,
         Producer_run_i,
         (Producer_run_i * Producer_run_i));
       Buffer_put(Producer_get_buffer(obj),
                  Producer_run_ad);
       Producer_run_i = (Producer_run_i + 1);
    :: else -> break
    od;
    Buffer_halt(Producer_get_buffer(obj));
  }
  unless {get_this_EXN > 0}
}
```

**Fig. 17.** The `Producer run` method

```
#define Producer_constr(obj,b)
  Producer_set_buffer(obj,b);
  run Producer_Thread(obj);
```

**Fig. 18.** The `Producer` constructor

PROMELA's `unless` construct seems very closely related to an exception construct, except for the fact that it works *"outside in"* instead of *"inside out"*, the latter being the case for exceptions. As an example consider the JAVA statement (assuming some variable `x`):

```
try{
  try{throw(new Exception());}
  catch(Exception e){x = 1;}
}
catch(Exception e){x = 2;}
```

The result of executing this statement should be that `x` is assigned to `1`, hence (only) the inner `catch` is invoked when the exception is thrown. In contrast, consider a related PROMELA statement, where now an `EXN` variable (initially 0) has been introduced. When this variable becomes positive it is regarded as if an exception has been thrown:

```
{
  {EXN = 1}
  unless {EXN > 0 -> x = 1}
}
unless {EXN > 0 -> x = 2}
```

The effect of this statement will be that `x` is assigned to 2. Hence, the outermost `unless` construct is invoked when `EXN` becomes positive. Gerard Holzmann has been very helpful to us by implementing a `-J` option (J for JAVA) in the verifier that changes the semantics of PROMELA in such a way that `unless` constructs are interpreted "*inside out*". This still leaves a second issue: that as soon as the inner `unless` is chosen (executing the inner `EXN > 0`), then the outer is invoked (since `EXN > 0` is still true) and `x` gets the value 2 anyway. Hence, we have to prevent that kind of behavior. Note that the above PROMELA code needs some modifications to actually "work" because if it is the last statement in "`P`" within the statement "`P unless Q`" that makes "`Q`" executable, then "`Q`" will actually *not* get executed.

Recall that a JAVA exception is an object of one of the exception classes, either one of the built in classes or a user defined exception class. Hence, an exception may contain data and methods. The `throw(e)` statement takes an exception object reference `e` as argument, and in our model we translate that into a store of that object reference in a special variable, `EXN`, in the data area of the thread (potentially the main program) where it has been thrown. All the `unless` constructs in the thread will test on this variable to see if it becomes different from `null`. The data area for the `Consumer` class is shown in Fig. 19. There are corresponding macros for accessing this variable. There are macros `get_EXN(obj)` and `set_EXN(obj,value)` for accessing this variable of a thread identified by `obj`. In addition, the macros `get_this_EXN` and `set_this_EXN` will access the `EXN` variable of *this* thread.

For each exception class `E`, there is a predicate `exn_E` which evaluates to true if the `EXN` variable contains an exception object of that class or a subclass thereof. For example, Fig. 20 shows this predicate for the built in `Exception` class. Recall that the `HaltException` was defined as a subclass thereof, as is `InterruptedException`.

Exception throwing is mainly modeled by the macros shown in Fig. 21. The `dotted_throw` macro throws an exception at a particular object given as parameter. This allows one thread to throw an exception at another thread, causing that other thread's `EXN` variable to be set. The `throw` macro throws an exception to the object identified by the free variable `obj` which is *this* current thread object. Finally, calling "`T.stop()`" in a JAVA program corresponds to throwing the predefined `THREAD_DEATH` exception object to the `T` object.

We shall now explain how exceptions are caught. Consider a `try` statement of the form:

```
try T
catch(E1 e1) C1
catch(E2 e2) C2
...
catch(En en) Cn
finally F
```

A JAVA statement of this form is translated into a PROMELA statement of the following form using macros defined in Fig. 23, and where a prime (') after a block indicates its translated version:

```
{ try(T')
  unless {
    if
    :: catch(exn_E1,e1,C1')
    :: catch(exn_E2 & !exn_E1,e2,C2')
    ...
    :: catch(exn_En & !exn_E1 & !exn_E2 &
         ... & !exn_En-1,en,Cn')
    fi}
}
unless {finally(F')}
```

For example, the `try` construct in the `get` method in Fig. 2 is translated into the PROMELA code in Fig. 22. The general idea behind this translation is that the inner `unless` construct catches any exceptions which match any one of the exception predicates `exn_E1` ... `exn_En`, and that the outer `unless` construct models the `finally` construct in JAVA: the block `F` must be executed no matter what as the last thing. For example, in the case where for example `C1'` itself throws a new exception, the `F` still has to be executed before that exception can be thrown further up. Now, let us explain the macros in Fig. 23.

```
typedef Consumer_Class{
  ObjRef EXN;
  ObjRef buffer;
};
```

**Fig. 19.** The `Consumer` object type

```
#define exn_Exception
  (get_class(get_this_EXN) == Exception
  |exn_InterruptedException
  |exn_HaltException)
```

**Fig. 20.** The `exn_Exception` predicate

```
#define dotted_throw(obj,exn)
  set_EXN(obj,exn)

#define throw(exn)
  dotted_throw(obj,exn)

#define stop(obj)
  dotted_throw(obj,THREAD_DEATH)
```

**Fig. 21.** Operations for throwing exceptions

```
{ try(Buffer_wait(obj))
  unless {
    if
    :: catch(exn_InterruptedException,
         Buffer_get_ex,skip)
    fi};
}
unless {finally(skip)};
```

**Fig. 22.** Translation of the `try` statement in the `get` method

```
#define try(s)
  {s;exit_to_final;skip}

#define exit_to_final
  throw(EXIT)

#define catch(exn_E,x,s)
  catch_cond(exn_E,x) -> s;exit_to_final

#define catch_cond(exn_E,e)
  d_step{
    (exn_E)
     ->
    e = get_this_EXN;
    set_this_EXN(NO_EXN)
  }

#define finally(s)
  final_cond -> s;exit_final;skip

#define final_cond
  d_step{
    get_this_EXN > 0 ->
    if
    :: get_this_EXN == EXIT ->
         set_this_EXN(NO_EXN)
    :: else ->
         set_this_EXN(- get_this_EXN)
    fi}

#define exit_final
  set_this_EXN(- get_this_EXN)
```

**Fig. 23.** Operations for catching exceptions

The `try(s)` macro executes `s` and then throws the predefined `EXIT` object (`exit_to_final`). The `EXIT` exception is then caught by the `finally` macro (always inserted at the end) as we shall see. The `catch(exn_E,x,s)` macro tests whether the current value of the `EXN` variable satisfies the `exn_E` predicate ( `catch_cond(exn_E,x)`), in which case that branch is executed.

The `catch_cond(exn_E,e)` macro in an atomic move tests the predicate `exn_E` (hence is only executable when it is true, see Fig. 20 for one such predicate), and then stores the value of the exception in the local variable `e` to be accessible within the block to be executed. Note also how the `EXN` variable is zeroed to the predefined `NO_EXN` value in order to avoid outer `unless` constructs being triggered, as already discussed earlier on page 10.

The `finally(s)` macro defines when the outer `unless` construct should be triggered, namely when `final_cond` becomes executable. The `final_cond` macro is executable whenever there is some exception object reference in the `EXN` variable (it is bigger than 0). Note that we reset this to 0 (`NO_EXN`) whenever an exception is caught at an inner level so that this outer final construct will not get activated. If there is an exception, and this is just a normal `EXIT` from one of the blocks `T,C1,...,Cn` in the `try` statement, this is then forgotten (`set_this_EXN(NO_EXN)`). If on the other hand (`else`) it concerns an exception that has not been caught, and we need to remember it so that it can be re-thrown after the code in the `finally` construct has been executed. We also need to make sure that it does not activate `unless` constructs higher up during the execution of the `finally` code. One way to do this is to negate it since the `unless` constructs in the translation only react on positive values of `EXN`. When we then leave the `finally` construct, we negate it back to its positive value so that it can be "thrown" further up. Some of the macros contain `skip` statements. These are necessary in order to make it work due to the semantics of the `unless` construct.

In Figs. 15 and Fig. 16 the macros `a_finally(s)` and `d_finally(s)` are called. These are defined as respectively `atomic{finally(s)}` and `d_step{finally(s)}`.

The `d_finally(Buffer_unlock(obj))` call in Fig. 16 unlocks the object in case an exception has been thrown out of a synchronized method. The `d_step` ensures that the exception is not thrown further up during the unlocking (which is more economic than negating the exception) and the unlocking can be made *completely* atomic since there are no blocking statements.

The `a_finally(Buffer_lock(obj))` call in Fig. 15 locks an object after a call of `wait` has been interrupted. This locking must be as atomic *as possible*, but with the possibility of blocking since another thread may own the lock. Therefore a `d_step` cannot be used (it does not allow blocking).

## *3.6 Value returning methods*

A JAVA return statement can have one of two forms. Either it has the form "`return`" in case the method is *not* value returning, or the form "`return` *exp*" if the method *is* value returning. In either case, such a return statement has the same effect as throwing an exception that is caught by a "return exception handler" surrounding the body of the value returning method, and by any `finally` constructs on the "way up" to that. Figure 24 shows how the `get` method of the `Buffer` class translates with respect to value return.

We see that the body of the method is surrounded by an `unless` construct triggered by the `return_cond` predicate. This predicate, defined in Fig. 25, becomes executable when the `EXN` variable in the executing thread (the one that executes the value returning method) gets the value `RETURN` which is a predefined fixed object reference. This happens for example when the `return_with` macro is called with an argument denoting the value returned. The `return_without` models the return from a method that does not return a value.

The assignment "`RES[TOP] = e`" in the `return_with` macro needs some explanation. The problem we are faced with is that JAVA allows value denoting expressions to contain calls of value returning methods that may have side effects. In PROMELA expressions are of a syntactically different class than statements, and cannot have side effects. Consider for example the following statement in the `Consumer` class's `run` method:

```
received[count] = (AttrData)buffer.get();
```

Ignoring the casting "`(AttrData)...`", the right hand side of this assignment has side effects, whereas the right hand side of a PROMELA assignment does not allow this. Our solution is to execute the method before the assignment, and then store the result value in a *result array* that is introduced for this purpose. Note that it has to be an array in order to model assignment statements of the form "`x = o.m(x) + p.n(y)`" where several value returning method calls occur in an expression. Part of the translation of the `Consumer` class's `run` method is shown in Fig. 26. The `Buffer_get` macro will execute and finally store the return value in `RES[TOP]`, where `RES` is the result array. This value is then "picked up" in the assignment statement. Note that if there are several value returning method calls in a single expression, the `TOP` variable is incremented for each during their evaluation, and then reset to its previous value before the values are accessed.

### *3.7 The `main` method*

The `main` method of the `Main` class is translated into the PROMELA `init` section as can be seen in Fig. 27. We have already explained the translation of object creation, which is the only activity of this particular `main` method. Recall that the constructors of the `Producer` and `Consumer` classes in this present example start the processes.

```
#define Buffer_get_code(obj)
  {
    ...
    return_with(Buffer_get_x);
  }
  unless {return_cond}
```

**Fig. 24.** Translation of the value returning `get` method

```
#define return_with(e)
  RES[TOP] = e;
  throw(RETURN)

#define return_without
  throw(RETURN)

#define return_cond
  d_step{
    get_this_EXN == RETURN ->
    set_this_EXN(NO_EXN)
  }
```

**Fig. 25.** Operations for return exceptions

```
#define MAX_RESULT 5

proctype Consumer_Thread(ObjRef obj){
  ...
  int RES[MAX_RESULT];
  byte TOP;
  ...
  Buffer_get(Consumer_get_buffer(obj));
  Consumer_run_received[Consumer_run_count] =
    RES[TOP];
  ...
}
```

**Fig. 26.** The `Consumer` thread

```
init{
  {ObjRef Main_main_b;
   ObjRef Main_main_p;
   ObjRef Main_main_c;
   atomic{
     ObjRef obj = create_object(Main,Main_Next);
     Main_Next++};
   atomic{
     Main_main_b =
       create_object(Buffer,Buffer_Next);
     Buffer_Next++};
   Buffer_constr(Main_main_b);
   atomic{
     Main_main_p =
       create_object(Producer,Producer_Next);
     Producer_Next++};
   Producer_constr(Main_main_p,Main_main_b);
   atomic{
     Main_main_c =
       create_object(Consumer,Consumer_Next);
     Consumer_Next++};
   Consumer_constr(Main_main_c,Main_main_b);
  } unless {get_this_EXN > 0}
}
```

**Fig. 27.** The `main` method

## 4 Analyzing the program

The presented program is correct in the sense that no errors are found by SPIN when applied to the PROMELA code generated by JPF. In order to illustrate the effectiveness of JPF (and SPIN of course) we have seeded 21 errors in the program shown in Fig. 2, and for each error analyzed the now incorrect program using JPF. This experiment is described in the following section.

### *4.1 The result*

The results of this experiment are shown in Table 1. For each error we give the line numbers changed, referring to Fig. 2, and the new contents of these lines. As an example, error 1 is obtained by changing line 3 to "`protected int putPtr = 1;`" (initializing to 1 instead of to 0).

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 deterministic. This means that executing a multi-threaded program several times 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 s. The memory consumption was around 17 Mb. This amount of memory must probably be explained by the modeling of the JAVA Virtual Machine within PROMELA.

"Errors" 11 and 20 are special (marked with a $*$) in the sense that they are not really errors when using the environment described in Sect. 2.2. This environment only creates one consumer, and to make the errors manifest themselves, we needed to create two consumers. In addition, with two consumers the assertions make no sense and were deleted. Hence, we were now 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 min and 77 Mb, 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 min, 28 Mb, and exploring 423096 states and 1 million transitions.

SPIN used 5.2 s to verify that the original program contained no errors. This involved the exploration of 33683 states and 61944 transitions, and a memory consumption of 18.8 Mb.

### *4.2 Comments on the result*

The example is small since its main purpose has been to illustrate the translation done by JPF. However, the exercise does show that around half (11) of the errors are not guaranteed to be caught when plainly executing the program on the newest version of JVM, while in contrast JPF finds the errors each time if present in the setup (errors 11 and 20 were not present with only one consumer). The difference is most obvious for errors 11, 14 and 20. Errors 3 and 11 were suggested by a reviewer. The chance of catching errors by executing the program gets even smaller when the program size increases.

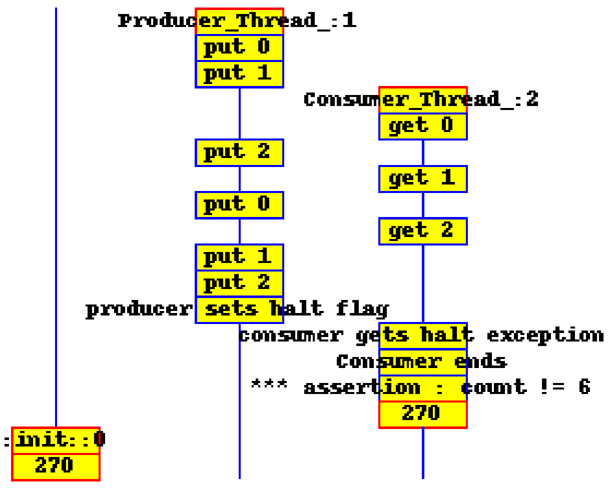

**Fig. 28.** Trace for error 14

**Table 1.** Verification results

| Nr. | Line | Modification (changed to) | JPF | JVM 1.1.6 | JVM 1.1.3 |
|---|---|---|---|---|---|
| 1 | 3 | `protected int putPtr = 1;` | A2 | 100 | 100 |
| 2 | 5 | `protected int usedSlots = 1;` | A1, A2 | 100 | 100 |
| 3 | 9 | `while (usedSlots != SIZE)` | D | 100 | 100 |
| 4 | 9 | `while (usedSlots == 2)` | D | 65 | 0 |
| 5 | 12<br>13 | `putPtr = (putPtr + 1) % SIZE;`<br>`array[putPrt] = x;` | A2 | 100 | 100 |
| 6 | 13 | `putPtr = putPtr % SIZE;` | A2 | 100 | 100 |
| 7 | 13<br>28 | `putPtr = (putPtr + 1) % 2;`<br>`getPtr = (getPtr + 1) % 2;` | A2 | 56 | 0 |
| 8 | 14 | `if (usedSlots == SIZE) notifyAll();` | D | 33 | 100 |
| 9 | 14 | **remove:**<br>*if (usedSlots == 0) notifyAll();* | D | 55 | 100 |
| 10 | 14<br>15 | `usedSlots++;`<br>`if (usedSlots == 0) notifyAll();` | D | 35 | 100 |
| 11* | 14<br>29 | `if (usedSlots == 0) notify();`<br>`if (usedSlots == SIZE) notify();` | D | 2 | 100 |
| 12 | 20 | `while (usedSlots == 0)` | D | 100 | 100 |
| 13 | 23 | `if (usedSlots != 0) {` | A1, D | 100 | 100 |
| 14 | 23 | `if (halted) {` | A1 | 3 | 0 |
| 15 | 29 | `if (usedSlots == 0) notifyAll();` | D | 50 | 0 |
| 16 | 29 | **remove:**<br>*(if usedSlots == SIZE) notifyAll();* | D | 44 | 0 |
| 17 | 29<br>30 | `usedSlots--;`<br>`if (usedSlots == SIZE) notifyAll();` | D | 66 | 0 |
| 18 | 30 | `usedSlots++` | A1, A2 | 100 | 100 |
| 19 | 35 | **remove:**<br>*halted = true;* | D | 100 | 100 |
| 20* | 36 | `notify();` | D | 2 | 100 |
| 21 | 36 | **remove:**<br>*notifyAll();* | D | 100 | 100 |

## *4.3 Error traces*

When SPIN identifies an error in the translated PROMELA code, be it a deadlock or an assertion violation, it returns with an error trace showing the sequence of executed PROMELA statements leading from the initial state to the state where the error occurs. Our tool does not currently map such PROMELA error traces back to corresponding JAVA traces. As an alternative, we have extended the `Verify` class with special `print` methods which can be called (by the programmer) in selected places in the JAVA code. These calls will then be translated into PROMELA print statements set up to print on a graphical two-dimensional *message sequence chart.* Figure 28 shows such a chart illustrating the trace for error number 14.

To obtain this chart, explained below, we have added print statements in 5 places in the code. For example, in lines 12 and 26 (pushing current lines downwards) we have added the statements:

```
12Verify.print("put",putPtr);
26Verify.print("get",getPtr);
```

Hence, what will be printed out is a text string (`"put"` or `"get"`), and the position in the buffer where a value is stored (put), retrieved from (get), respectively. Other print statements record other events, such as the producer setting the `halted` flag, and the consumer getting the corresponding exception. Figure 28 illustrates a situation where the producer (center vertical line) first produces two values, stored in positions 0 and 1. Then the consumer gets the value in position 0, etc. The consumer only gets the first three values. The producer then finishes and sets the `halted` flag, where after the consumer now gets an exception when trying to get the fourth value (in position 0). Therefore the consumer will miss three values, and the first assertion (`count == 6`) will be broken. Note that the second assertion is not broken.

Adding these print statements requires thought and work, but the idea may in fact be useful to cut down the enormous amount of information contained in the error traces created by Spin.

## 5 Conclusion and future work

In this paper we have described the initial prototype of a translator from a non-trivial subset of the general purpose programming language Java to the model checking language Promela. We have also applied the translator to 21 bugged variations of a small example program, observing that the model checker catches all the bugs, while executing the programs only catch the bugs safely in half of the cases.

We are currently applying the translator to a collection of real programs developed within NASA, and the outcome of these experiments will be published at a later time. As an example, a collaboration with NASA's Goddard Space Center involves the verification of a satellite file down-link protocol written in Java. The translator has in addition been applied to analyze a Chinese Chess game server application written in Java. This leads to the confirmation of a suspected deadlock and the identification of a smaller scenario leading to that deadlock. This work is documented in [11].

An essential question is whether the translation is optimal. We still need to evaluate this question. There may be other ways to translate into Promela and one may consider making a by-pass, avoiding Promela, and translating directly into the C interface to the Spin model checker. A major point of discussion has been how to model object creation. One early thought was to use Promela's proctypes to model classes, and then spawn a process for each object creation. However, this solution turned out to be less attractive since one has no direct access to the variables of a process from outside the process. An extension of Spin allowing such accesses is described in [15]. Recently Jpf has been modified to translate object synchronizations and exceptions more efficiently by introducing two new special purpose arrays holding locking and exception variables, thus keeping these variables separate from user defined variables. This gave a 50% reduction in the amount of C-code generated by Spin. We have presented the simpler solution for readability reasons.

We see the current translator as a prototype experiment (involving 9 man months until now). Our plans for the future involve writing a model checker directly for Java. What is equally important is our planned efforts to build an abstraction workbench around Java, where programs can be reduced in size before model checking is applied. Techniques such as program slicing, abstract interpretation, and partial evaluation will have a great influence.

We believe that the kind of technology presented in this paper, already as is, can be very useful for unit testing where one focuses on a single or a few classes, just as has been demonstrated with the example. This requires setting up an aggressive environment consisting of a collection of threads which will "bombard" the unit with accesses. Finally, we believe that perhaps the technology can be useful for students learning to program in Java.

Concerning the specification language, our main approach has been not to extend the Java language but to express temporal properties as calls to methods defined in a special temporal logic class (the `Verify` class), all of whose methods are static (hence one does not need to instantiate the class to objects before calling the methods). In addition to the `assert` method one can for example imagine an `always` method, an `eventually` method, and basically include all of Spin's linear temporal logic operators as methods, having Boolean return types in addition to Boolean argument types, so that they can be composed. Calls of such methods will then generate Ltl formulae to be verified, referring to the position where they are called in the code.

*Acknowledgements.*
We thank Gerard Holzmann for changing Spin to support the translation of exceptions. We thank the members of the Automated Software Engineering team at NASA Ames Research Center for their comments; and in particular Mike Lowry for supporting the project, and John Penix and Willem Visser for proof reading the manuscript. We also thank the reviewers. We have used a Java parser ported by Peter Sestoft (the Royal Veterinary and Agricultural University in Denmark) to Moscow ML from a Standard ML version written by Olivier Brunet and Gordon Woodhull in Alex Aiken's group (University of California, Berkeley, USA).

## References

1. Arnold, K., Gosling, J.: The Java Programming Language. Reading, MA: Addison Wesley, 1996
2. Cattel, T.: Modeling and verification of sC++ Applications. In: TACAS '98: Tools and Algorithms for the Construction and Analysis of Systems, Lisbon, Portugal. LNCS 1384. Berlin, Heidelberg, New York: Springer-Verlag, 1998
3. Corbett, J.C.: Constructing compact models of concurrent Java programs. In: Proc. ACM Sigsoft Symposium on Software Testing and Analysis, Clearwater Beach, FL, USA, 1998
4. Demartini, C., Iosif, R., Sisto, R.: Modeling and validation of Java multithreading applications using SPIN. In: Proc: 4th SPIN Workshop, November 1998, Paris, France
5. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, CA, USA, 1998
6. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Reading, MA: Addison Wesley, 1996
7. Havelund, K.: Mechanical verification of a garbage collector. In: Méry, D., Sanders, B. (eds.): FMPPTA '99: 4th Int. Workshop on Formal Methods for Parallel Programming: Theory and Applications, San Juan, PR, USA. LNCS 1586. Berlin,

Heidelberg, New York: Springer-Verlag, 1999

8. Havelund, K., Larsen, K.G., Skou, A.: Formal verification of an audio/video power controller using the real-time model checker UPPAAL. In: 5th Int. AMAST Workshop on Real-Time and Probabilistic Systems, Bamberg, Germany. LNCS 1601. Berlin, Heidelberg, New York: Springer-Verlag, 1999
9. Havelund, K., Lowry, M., Penix, J.: Formal Analysis of a Space Craft Controller using SPIN. In: Proc. 4th SPIN Workshop, November 1998, Paris, France. IEEE Trans. on Software Engineering, (to appear)
10. Havelund, K., Shankar, N.: Experiments in Theorem Proving and Model Checking for Protocol Verification. In: Gaudel, M.-C., Woodcock, J. (eds.): FME '96: Industrial Benefit and Advances in Formal Methods, Oxford, UK. LNCS 1051. Berlin, Heidelberg, New York: Springer-Verlag, 1996
11. Havelund, K., Skakkebæk, J.: Applying model checking in Java verification. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.): Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, July and September 1999, Trento, Italy – Toulouse, France (presented at the 6th Workshop). Describes an application of JPF to a game server. LNCS 1680. Berlin, Heidelberg, New York: Springer-Verlag, 1999
12. Havelund, K., Skou, A., Larsen, K.G., Lund, K.: Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In: Proc. 18th IEEE Real-Time Systems Symposium, San Francisco, CA, USA, 1997
13. Holzmann, G.: The Design and Validation of Computer Protocols. Englewood Cliffs, MA: Prentice Hall, 1991
14. Pell, B., Gat, E., Keesing, R., Muscettola, N., Smith, B.: Plan Execution for Autonomous Spacecrafts. In: Proc. Int. Joint Conference on Artificial Intelligence, August 1997, Nagoya, Japan
15. Visser, W., Havelund, K., Penix, J.: Adding Active Objects to SPIN. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.): Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, July and September 1999, Trento, Italy – Toulouse, France (presented at the 5th Workshop). LNCS 1680. Berlin, Heidelberg, New York: Springer-Verlag, 1999