Translating Java to Spin a step towards the JavaProver

Klaus Havelund and* Thomas Pressburger

NASA Ames Research Center, Recom Technologies, Moffett Field, California, USA.
Email: {havelund,ttp}@ptolemy.arc.nasa.gov

Abstract. This report outlines some high level ideas for translating Java to the Promela, the "programming language" of the Spin model checker. The purpose is to establish a framework for verification and debugging of Java programs based on model checking. This work shall be seen in a broader attempt to make formal methods applicable "in the loop" of programming within NASA's areas such as space, aviation, robotics, and ground control software. Our main goal is to achieve automated formal methods, such that programmers themselves can apply these in their daily work (in the loop) without specialists interfering and without necessarily having to manually reformulate a program in a different notation in order to analyze the program. The work has been spawned of by an effort to formally verify, using Spin, a multi–threaded operating system programmed in Lisp for space crafts. A Java Specification language is furthermore presented, which makes it possible to state properties about Java programs. The language is new in the sense that it is based on linear temporal logic formulae that can occur any place a statement can occur. Hence, these formulae can be deeply embedded in Java code and should correspondingly be interpretated on paths only that start at these points – and not from the beginning of the program execution as is the case for Spin Ltl formulae.

* With help from Michael Lowry and Mahadevan Subraniam

Table of Contents

1 Introduction 4
- 1.1 Source and Target Language 4
- 1.2 Motivation for the Work 5
- 1.3 Outline of the Report 5

2 Extensions to the SPIN Language 7

3 Classes, Objects and Methods 8
- 3.1 Example JAVA Program to be Translated 8
- 3.2 Translation 9
  - 3.2.1 General Ideas 9
  - 3.2.2 Translation of Class `X` 10
  - 3.2.3 Translation of Class `XY` 11
  - 3.2.4 Translation of Class `Main` 13

4 Threads 14
- 4.1 Example JAVA Program to be Translated 14
- 4.2 Translation 15
  - 4.2.1 Modifying Type `ClassName` 15
  - 4.2.2 Translation of Class `Adder` 15
  - 4.2.3 Translation of Class `Main` 15

5 Synchronization 17
- 5.1 Example JAVA Program to be Translated 17
- 5.2 Translation 17
  - 5.2.1 General Ideas 17
  - 5.2.2 Translation of Class `XY` 19

6 Wait, Notify and NotifyAll 21
- 6.1 Example JAVA Program to be Translated 21
- 6.2 Translation 23

7 Suspend and Resume 27
- 7.1 Example JAVA Program to be Translated 27
  - 7.1.1 More about Specifying Properties 27
  - 7.1.2 The JAVA Program 28
  - 7.1.3 More About the Specification 30

7.2 Translation 30
7.2.1 General Ideas 30
7.2.2 Translation of Class `Status` 32
7.2.3 Translation of Class `Agent1` 32
7.2.4 Translation of Class `Agent2` 33
7.2.5 Translation of Class `Main` 34
7.3 Alternative Translations 35

8 Stop, Join, isAlive 36
8.1 Example JAVA Program to be Translated 36
8.1.1 The JAVA program 36
8.1.2 The Specification 38
8.2 Translation 38
8.2.1 General Ideas 38
8.2.2 Translation of Class `Obj` 42
8.2.3 Translation of Class `Data` 42
8.2.4 Translation of Class `Job` 44
8.2.5 Translation of Class `Stop` 44
8.2.6 Translation of Class `Main` 45

9 Yield and Sleep 47

10 Data Structures 48

11 Expressions 49

12 Exceptions 50

13 Priorities 51

14 Specifications 52
14.1 The `Verify` Class 52
14.1.1 Syntax and Outline of Semantics 52
14.1.2 More about Semantics 52
14.1.3 Correspondence to LTL 55
14.2 Interpretation of Verifications 56
14.3 Using Inheritance to Avoid Class Prefixes 57
14.4 Properties as Methods instead of as Statements 59
14.5 Comments and Language Extension 60

A The Producer/Consumer Example 62

1 Introduction

This report outlines some high level ideas for translating JAVA to the PROMELA, the "programming language" of the SPIN model checker. The purpose is to establish a framework for verification and debugging of JAVA programs based on model checking. This work shall be seen in a broader attempt to make formal methods applicable "in the loop" of programming within NASA's areas such as space, aviation, robotics, and ground control software. Our main long term goal is to achieve an automated formal methods workbench, JAVAPROVER, for JAVA programming, such that programmers themselves can apply these in their daily work (in the loop) without specialists interfering and without necessarily having to manually reformulate a program in a different notation in order to analyze it. Although the presented work is based on a translation of JAVA to PROMELA, future work will very likely focus on writing a model checker for JAVA from scratch, thereby obtaining a more efficient system. The work presented in this report will provide an initial pioneering effort along that line.

1.1 Source and Target Language

JAVA [1] is a general purpose object oriented programming language with built in mechanisms for multi-threaded programming [5]. It was originally designed to support internet programming, but goes well beyond this domain. It is always difficult to predict the future, but a prediction is that JAVA becomes as widely spread as is C++ today. JAVA is a relatively simple language compared to C++, and it is regarded as a much safer language, amongst other things due to its automatic garbage collection and lack of general pointers. In spite of its simplicity it appears to be a powerful language.

SPIN [4] 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. Processes communicate either via shared variables or via message passing through buffered channels. Properties to be verified are stated 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, an error trace is generated. For the remaining part of this report, we shall often use the term SPIN to also represent the PROMELA language.

There are several reasons for choosing JAVA as the source language. First of all it seems to become a popular language that will attract a lot of attention from the programming community. There are good chances that JAVA will become widely used. As a well-known Computer Scientist (who probably wants to stay anonymous) said: "*in a few years, 999 out of 1000 programs will be written in* JAVA". Second, it is object oriented, which seems to be a strong trend in programming language design. Object orientation has strangely enough not yet hit the model checking community. Certain attempts have been made to provide object oriented versions of pure specification languages such as Z (OOZ) and VDM (VDM++). Hence, proving a model checker for an object oriented language appears to be a research topic. Third, and finally, JAVA is quite a simple language, and in fact not that far away from PROMELA in complexity. It seems to be a natural "*next step*".

It is often claimed that model checkers of today cannot handle real programs, and consequently neither real JAVA programs. This is certainly true. However, there are two 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, JAVA can be used as a design language, just as PROMELA.

1.2 Motivation for the Work

The work has been spawned of by an effort to formally verify, using SPIN, a multi–threaded operating system for space crafts. This work is documented in [3]. The operating system is one component of NASA's New Millennium Remote Agent (RA) [6], an artificial intelligence based spacecraft control system architecture that is scheduled to launch in December of 1998 as part of the DEEP SPACE 1 mission to Mars. 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.

The hardest part of that work was to understand another persons code and to translate from this code to PROMELA. A third activity was to perform abstractions before the translation. That is, basically throwing away code irrelevant to the properties we wanted to verify. The conclusion from this work was that the programmer him/her self should carry out the verification (solves the understanding problem) and that the translation should be automatic. What is left is the abstraction problem, and for that purpose we have a parallel effort to build an *abstraction workbench* where one can directly reduce a JAVA program's state space for the purpose of efficient modelchecking.

1.3 Outline of the Report

Section 2 presents a few extensions to the PROMELA language that we have found useful for the presentation. Then follows a sequence of sections, 2–13, each discussing the translation of a particular JAVA concept. As a general approach, in each section we present the translation of a complete JAVA program that illustrates the concepts focused on in that section. To a certain extend, the report serves two purposes: to define the translation, and to show how (whether) it works. In other words, the report documents a sequence of experiments with hand translating Java programs. One of the translations has a size that made it qualify as being included as an appendix.

During the work a language for specifying properties about Java programs has developed on the sideline. Most of the Java programs presented in this report include such specifications, and a final section gives a general outline of such a specification language. This language is new in the sense that it is based on linear temporal logic formulae that can occur any place a statement can occur. Hence, these formulae can be deeply embedded in Java code and should correspondingly be interpretated on paths only that start at these points (and not from the beginning of the program execution as is the case for Spin Ltl formulae). Of course they will be translated to Spin's Ltl formulae.

The report documents work in progress, and does currently not cover all of Java.

2 Extensions to the Spin Language

We shall use the following notation for enumerated types:

```
type
  process_state = {suspended,waiting,runnable,stopped}
```

This will represent the following macro definitions:

```
#define process_state byte
#define suspended 0
#define waiting   1
#define runnable  2
#define stopped   3
```

Also, we shall assume we can write type equations of the form:

```
type
  array_index = byte
```

being mapped into:

```
#define array_index byte
```

3 Classes, Objects and Methods

The presentation of the translation scheme will be illustrated by translation of real complete JAVA programs. In this way we hope to provide a more complete comprehension of the translations needed, to ourselves as well as to the reader.

3.1 Example JAVA Program to be Translated

A JAVA class is generally speaking described by a name, a set of data variables, a constructor (method) that is executed when an object is created, and a collection of methods. Consider the complete JAVA program in figure 1, which we shall translate into SPIN.

This program defines first of all a class `Verify`, which basically just defines the method `assert`, which when called with a string "t" and a false argument prints: `assertion t broken`. The method is defined as static meaning that the method can be called without creating an object of the class. We shall call this method in places where we want an assertion to be verified. Running the above program will in fact **not** result in this assertion to be broken. Hence, the program is correct with respect to that assertion, and the corresponding SPIN program should therefore **neither** detect an error.

The class `X` introduces a variable `x` and a method for updating it. Class `XY` extends `X` by adding another variable `y`, a method for updating that, and a method `add` for updating `x` as well as `y`. The essential contents of this method is the calls:

```
add2x(dx);
add2y(dy);
```

The code surrounding these two statements makes up a *post condition*:

```
int old_x = x;
int old_y = y;

...

Verify.assert("updated", x == old_x + dx &
                         y == old_y + dy);
```

That is, the auxiliary variables `old_x` and `old_y` are introduced to hold the *old* values of the updated variables.

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

class X{
  public int x;
  public X(){x = 0;}
  public void add2x(int d){x = x + d;}
}

class XY extends X{
  public int y;
  public XY(){y = 0;}
  public void add2y(int d){y = y + d;};
  public void add(int dx, int dy){
    int old_x = x;
    int old_y = y;
    add2x(dx);
    add2y(dy);
    Verify.assert("updated", x == old_x + dx &
                             y == old_y + dy);
  }
}

class Main{
  public static void main(String[] args){
    XY xy = new XY();
    xy.add(4,4);
  }
}
```

Fig. 1. JAVA : An example program

3.2 Translation

3.2.1 General Ideas 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 in fact not assume any garbage collection at all. 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). Methods are mapped into macros.

Before we go the translation of each class, a few types needs to be defined, see figure 2.

```
type
  Index = byte,
  ClassName = {X,XY,Main}
typedef
  ObjRef{ClassName class; Index index};
```

Fig. 2. SPIN : Some basic types

The type `ClassName` is an enumerated type containing the names of all classes defined in the JAVA program. The type `ObjRef` represents object references: an object reference is a pair consisting of a class, and an index pointing to the position in the array holding the corresponding data area. Two more constants are needed, as defined in figure 3.

```
#define undefined 0
#define MAX 5
```

Fig. 3. SPIN : A couple of constants

The constant `undefined` is used to denote a "don't care" value used in particular contexts. The constant `MAX` denotes the maximum number of objects allocated for each class, hence the size of the arrays. Now to the translation of the three classes. We translate class `X` first, as shown in figure 4.

3.2.2 Translation of Class `X` First, a `typedef` declaration defines the type `X_Class` of `X`'s data area, just containing a single integer variable `x`. An array `X_Obj` of these records is defined together with a variable `X_Next`, that at any point in time points to the next free record to be allocated on a new object creation.

Then follows a couple of macros `X_get_x` and `X_set_x` for accessing the variable `x` (there are two such macros for each variable). These macros take as parameter an object identification of type `ObjRef`. The idea here is, that the methods call these two macros for accessing the variable `x`. As can be seen from

```
typedef X_Class{
  int x
};
X_Class X_Obj[MAX];
Index X_Next = 0;

#define X_get_x(obj)
  (obj.class == X  -> X_Obj[obj.index].x :
   (obj.class == XY -> XY_Obj[obj.index].x : undefined))

#define X_set_x(obj,value)
  if
  :: obj.class == X  -> X_Obj[obj.index].x = value
  :: obj.class == XY -> XY_Obj[obj.index].x = value
  fi

#define X_constr(obj)
  ObjRef obj;
  obj.class = X;
  atomic{obj.index = X_Next; X_Next++};
  X_set_x(obj,0)

#define X_add2x(obj,d)
  X_set_x(obj,X_get_x(obj) + d)
```

Fig. 4. SPIN : Translation of class `X`

the JAVA program, the class `XY` is defined as a subclass of the class `X`. Hence, at runtime, in order to access the variable of an object, one needs to know which class (`X` or `XY`) in order to access the correct SPIN array, and of course also which index (which record in that array). There will be a conditioned branch for each subclass of the class where the variable is introduced.

The macro `X_constr` defines the constructor. The JAVA constructor is parameter less, while the SPIN constructor takes an object as parameter, and "allocates" this object. Hence, the JAVA declaration: "`X x = new X()`" would correspond to the macro call "`X_constr(x)`". One sees how the class is set to `X` and how the index is set to point to the next free record. The variable `x` is then initialized to 0.

Finally, the method `add2x` is translated into the macro `X_add2x`. Note how all macros are parameterized with object references of the type `ObjRef`.

3.2.3 Translation of Class `XY` The translation of the class `XY` follows pretty much the same rules, and is shown in figure 5. The type `XY_Class` now contains the variable `x` from class `X` (which class `XY` extends), and the variable `y` which is new. This is how we basically model inheritance: variables from inherited classes are simply included (simple text inclusion) in the resulting record. Concerning the variable accessing macros, note that these have no conditional bodies since the class XY has no further subclasses in the program.

```
typedef XY_Class{
  int x;
  int y
};
XY_Class XY_Obj[MAX];
Index XY_Next = 0;

#define XY_get_y(obj)
  XY_Obj[obj.index].y

#define XY_set_y(obj,value)
  XY_Obj[obj.index].y = value

#define XY_constr(obj)
  ObjRef obj;
  obj.class = XY;
  atomic{obj.index = XY_Next; XY_Next++};
  X_set_x(obj,0);
  XY_set_y(obj,0)

#define XY_add2y(obj,d)
  XY_set_y(obj,XY_get_y(obj) + d)

#define XY_add(obj,dx,dy)
  old_x = X_get_x(obj);
  old_y = XY_get_y(obj);
  X_add2x(obj,dx);
  XY_add2y(obj,dy);
  assert( X_get_x(obj) == old_x + dx &
          XY_get_y(obj) == old_y + dy)
```

Fig. 5. SPIN : Translation of class XY

The constructor `XY_constr` allocates an `XY` object and initializes all variables: the inherited `x` (by repeating class `X`'s constructor contents) and the new `y`. Finally the two new methods are translated into two macros.

The example shows how we deal with variables local to methods. The method `add` has two local variables: `old_x` and `old_y`. Since SPIN does not have a local scope concept, except for proctypes, we have to declare these two variables globally to the thread that finally calls this method, and in this case it is the main program. The main program can be thought of as a thread in its own. See figure 6 below. One can think of this way to deal with variables local to methods as the way such variables (local to SUBROUTINEs) are laid out in FORTRAN programs. Each such variable is only laid out once in a static area. This of course prevents recursion.

We see how the `assert` statement of SPIN verifies the condition corresponding to the `Verify.assert` call in the JAVA program. This is a way to verify post conditions.

3.2.4 Translation of Class `Main`

The class `Main` is translated into SPIN's `init`–section, see figure 6. We assume in general that a `Main` class is defined, which only contains one static method called `main`. We see how an object of type `XY` is constructed, and how one of its methods is called.

```
init{
  int old_x;
  int old_y;
  XY_constr(xy);
  XY_add(xy,4,4);
}
```

Fig. 6. SPIN : Translation of class `Main`

4 Threads

Threads in JAVA allow us to write concurrent programs. Each JAVA thread corresponds to a process (in SPIN terminology), and threads therefore execute in parallel.

4.1 Example JAVA Program to be Translated

Consider the JAVA program in figure 7. The classes `Verify`, `X` and `XY` are as before. We have added the class `Adder`, which extends the `Thread` class: hence, objects of this class are threads. A thread class must contain a `run` method which is executed when an object of that class is started with the `start` method (defined in the system–defined `Thread` class). The program works as follows: two threads of the `Adder` class are started, each working one the same `XY` object. Each thread adds 4 to `x` as well as to `y` of the shared `XY` object (given as parameter to the thread constructors). This way of sharing an object between two or more threads is common in JAVA.

```
class Verify{...};
class X{...};
class XY extends X{...};

class Adder extends Thread{
  private XY xy;

  public Adder(XY xy){this.xy = xy;}

  public void run(){xy.add(4,4);}
}

class Main{
  public static void main(String[] args){
    XY xy = new XY();
    Adder adder1 = new Adder(xy);
    Adder adder2 = new Adder(xy);
    adder1.start();
    adder2.start();
  }
}
```

Fig. 7. JAVA : Example extended with threads

In this new program, the assertion in the `add` method in the `XY` class no longer holds in all reachable state: the post condition for the method is *false.*

The assertion postulates that after each call of `add` (before it terminates) the new values of `x` and `y` are as the old values, except for updated with the new deltas. They are not, however, if threads are arbitrarily interleaved: suppose one thread executes `xy.add(4,4)` and now wants to execute the assertion. Suppose then that the other thread, before the assertion, executes, hence changing the variables. Now the assertion of the first thread is not true. When we run the JAVA program, this will not be discovered. When running the SPIN model this error is identified immediately. The result is an error trace illustrating the above described situation.

4.2 Translation

4.2.1 Modifying Type `ClassName` Before we translate the new classes, the type `ClassName` has to be modified to include the new class name `Adder`, as shown in figure 8.

```
type ClassName = {X,XY,Adder,Main}
```

Fig. 8. SPIN : Extending the `ClassName` type

4.2.2 Translation of Class `Adder` The translation of the `Adder` class is shown in figure 9. Since it is "just" a class, it has basically the same kinds of components as the other classes we have seen. That is, the array of data areas, the macros that access the variables, and the constructor. Note how the macro `Adder_set_xy` assignes the value in two sub-statements. This is necessary because SPIN does not allow assignment to record variables, except for field by field. What is new is the `Adder_Thread` proctype definition. That is, the `run` method of the JAVA program is translated into a SPIN proctype, which is then made an instance of for each time an object of this class is started with the `start` method. Note how the variable `old_x` and `old_y` local to the `add` method in the `XY` class are now "layed out" in the `Adder_Thread` proctype. Hence, when we later create several instances of this proctype, we will get as many layouts of these variables.

4.2.3 Translation of Class `Main` The `Main` class is (again) translated into an `init` section as shown in figure 10. We see how an object `xy` of the class `XY` is created, and how it is given as parameter to the two `Adder` threads. The `start` calls in the JAVA program are translated into `run`–statements.

```
typedef Adder_Class{
  ObjRef xy
};
Adder_Class Adder_Obj[MAX];
Index Adder_Next = 0;

#define Adder_get_xy(obj)
  Adder_Obj[obj.index].xy

#define Adder_set_xy(obj,value)
  Adder_Obj[obj.index].xy.class = value.class;
  Adder_Obj[obj.index].xy.index = value.index

#define Adder_constr(obj,xy)
  ObjRef obj;
  obj.class = Adder;
  atomic{obj.index = Adder_Next; Adder_Next++};
  Adder_set_xy(obj,xy)

proctype Adder_Thread(ObjRef obj){
  int old_x;
  int old_y;
  XY_add(Adder_get_xy(obj),4,4);
}
```

Fig. 9. SPIN : Translation of class `Adder`

```
init{
  XY_constr(xy);
  Adder_constr(adder1,xy);
  Adder_constr(adder2,xy);
  run Adder_Thread(adder1);
  run Adder_Thread(adder2)
}
```

Fig. 10. SPIN : Translation of class `Main`

5 Synchronization

5.1 Example JAVA Program to be Translated

Learning from the error trace produced by the modelchecker we decide to correct our program. Basically, we want to prevent two (or more) calls of the `add` method to execute concurrently – at the same time. One call should finish before the other is started. One way to obtain this is to declare the `add` method as synchronized, using the `synchronized` keyword of JAVA as illustrated in figure 11. In fact, here we have also declared `add2y` as synchronized, even though this is not necessary if we only want to prevent several concurrent calls of the `add` method. That is, all methods that a syncrhonized method calls will become synchronized also relative to that call. However, we want to demonstrate that our technique also works for the case where several methods that call each other are synchronized.

```
class XY extends X{
  public int y;
  public XY(){y = 0;}
  public synchronized void add2y(int d){y = y + d;};
  public synchronized void add(int dx, int dy){
    int old_x = x;
    int old_y = y;
    add2x(dx);
    add2y(dy);
    Verify.assert("updated", x == old_x + dx &
                             y == old_y + dy);
  }
}
```

Fig. 11. JAVA : methods `add2y` and `add` become synchronized

5.2 Translation

5.2.1 General Ideas Only one thread at a time may execute any "synchronized" method on the same object. This requires mutual exclusion. Our solution is to add an integer typed field `LOCK` to the data area for an object that has synchronized methods, which at any time contains the *process identifier* of the thread that currently is executing a synchronized method on the object. In case no synchronized method is executed, the `LOCK` field contains the `null` value being equal to -1 (proper process id's in SPIN are always non-negative). Hence, once this field is set to a proper process id by a process that calls a synchronized method, only the process with this process id is allowed to operate on the object. When the call of the synchronized method terminates, the lock is *released* by setting it back to `null`. These modifications only affect the translation of class `XY`, which is shown in figures 14 and 15 to be explained in a moment.

First of all, however, we need two extra constants, `null` and `this`, as defined in figure 12. The constant `null` denotes the non-proper process id, while the constant `this` at any time denotes the process id of the currently running process, in SPIN denoted by `_pid`.

```
#define null -1
#define this _pid
```

Fig. 12. SPIN : Process id constants

Figure 14 shows first part of the translation of class `XY`. It shows the new definition of the typedef `XY_Class` which now includes the new `LOCK` field. Since this field is just like any other variable, two macros `get_LOCK` and `set_LOCK` are introduced to access and modify it, see figure 13. Furthermore, two macros `lock` and `unlock` for locking and unlocking an object are introduced. The `lock` macro is called by a process (coresponding to a JAVA thread) when that process calls a synchronized method. The procedure locks the object by assigning the calling process's process id to the `LOCK` field of the object. The locking can, however, only be allowed if noone else has locked the object, hence only if the `LOCK` field has the value `null`. Unlocking of course then means resetting the value to `null`.

```
#define get_LOCK(obj)
  XY_Obj[obj.index].LOCK

#define set_LOCK(obj,value)
  XY_Obj[obj.index].LOCK = value

#define lock(obj)
  atomic{
    get_LOCK(obj) == null ->
      set_LOCK(obj,this)}

#define unlock(obj)
  set_LOCK(obj,null)
```

Fig. 13. SPIN : Synchronization locking

5.2.2 Translation of Class XY

Figure 14 shows first part of the translation of class XY. The constructor XY_constr is modifed to initialize the LOCK field to null.

```
typedef XY_Class{
  int LOCK;
  int x;
  int y
};
XY_Class XY_Obj[MAX];
Index XY_Next = 0;

#define XY_get_y(obj)
  XY_Obj[obj.index].y

#define XY_set_y(obj,value)
  XY_Obj[obj.index].y = value

#define XY_constr(obj)
  ObjRef obj;
  obj.class = XY;
  atomic{obj.index = XY_Next; XY_Next++};
  set_LOCK(obj,null);
  X_set_x(obj,0);
  XY_set_y(obj,0)
```

Fig. 14. SPIN : Translation of class XY – part I

Figure 15 shows second part of the translation of class XY. For each synchronized method M (add2y and add) in the JAVA program we introduce two macros: XY_M_code and XY_M. The XY_M_code macro contains the proper code of the method. The XY_M macro calls the "proper" macro, but performs locking before the call if needed, and consequently unlocking after. Locking only takes place if the LOCK field of the object differs from the process id of the process – identified by this – that wants to execute the macro. In case the LOCK field is null the locking then takes place and the "proper" macro" is called. In case some other process has locked the object, locking blocks until this other process has unlocked the object again. In case the LOCK field already contains the process id of the calling process, then the call just proceeds. This happens for example when the XY_add macro calls the XY_add2y macro.

```
#define XY_add2y_code(obj,d)
  XY_set_y(obj,XY_get_y(obj) + d)

#define XY_add2y(obj,d)
  if
  :: get_LOCK(obj) == this ->
       XY_add2y_code(obj,d)
  :: else ->
       lock(obj);
       XY_add2y_code(obj,d);
       unlock(obj)
  fi

#define XY_add_code(obj,dx,dy)
  old_x = X_get_x(obj);
  old_y = XY_get_y(obj);
  X_add2x(obj,dx);
  XY_add2y(obj,dy);
  assert( X_get_x(obj) == old_x + dx &
         XY_get_y(obj) == old_y + dy)

#define XY_add(obj,dx,dy)
  if
  :: get_LOCK(obj) == this ->
       XY_add_code(obj,dx,dy)
  :: else ->
       lock(obj);
       XY_add_code(obj,dx,dy);
       unlock(obj)
  fi
```

Fig. 15. Spin : Translation of class XY – part II

6 Wait, Notify and NotifyAll

Calling the `wait()` method within a synchronized method suspends the current thread and allows other threads to execute synchronized methods on the object. Calling the `notify()` method allows one (arbitrarily chosen) suspended thread to continue past the `wait()`. There is also a `notifyAll()` method that wakes up all threads currently having executed a `wait()`.

6.1 Example Java Program to be Translated

The Java program, see figures 16 and 17, that we shall translate applies all the techniques that we have introduced so far, plus the `wait()` and `notify()` methods. The program consists of a `Producer` and a `Consumer` that communicates through a shared data structure, the `CubbyHole`. The producer generates the numbers from 0 to 9 and stores them in the cubbyhole, while the consumer reads these numbers. In order to verify the correctness of the program, we have introduced an array of `received` numbers within the `run` method of the `Consumer`. This array is updated for each received value, and a final loop verifies (calling the `assert` method) that the received numbers are indeed those from 0 to 9.

Note that neither the producer, nor the consumer makes any effort to ensure that the consumer is getting each value produced by the producer once and only once. The synchronization between these two threads actually occurs at a lower level, within the synchronized `put` and `get` methods of the `CubbyHole` object. These methods call the `wait()` and `notify()` methods in order to ensure that the producer does not produce numbers quicker than the consumer can consume them, which would cause numbers to be lost. Dually, it is also ensured that the consumer does not consume numbers faster than the producer can produce them, which would cause single values to be consumed more than once. We want the consumer to `get` each integer `put` by the producer exactly once.

The `CubbyHole` class contains two private variables: `contents`, which contains the value produced at any moment, and `available`, which is *true* whenever a value has been produced that has not yet been consumed. This variable is used for the correct synchronization between the `Producer` and `Consumer`.

```
class Producer extends Thread{
  private CubbyHole cubbyhole;
  public Producer(CubbyHole c){cubbyhole = c;}

  public void run(){
    for (int i = 0; i < 10; i++){
      cubbyhole.put(i);}
  }
}

class Consumer extends Thread{
  private CubbyHole cubbyhole;
  public Consumer(CubbyHole c){cubbyhole = c;}

  public void run(){
    int value = 0;
    int[] received = new int[10];
    for (int i = 0; i < 10; i++){
      value = cubbyhole.get();
      received[i] = value;};
    for (int i = 0; i < 10; i++){
      Verify.assert("received ok",received[i] == i);};
  }
}

class CubbyHole{
  private int contents;
  private boolean available = false;

  public synchronized int get(){
    while (available == false){
      try{wait();} catch (InterruptedException e) {}};
    available = false;
    notify();
    return contents;
  }

  public synchronized void put(int value){
    while (available == true){
      try{wait();} catch (InterruptedException e) {}};
    contents = value;
    available = true;
    notify();
  }
}
```

Fig. 16. JAVA : An example program – part I

```
class Main{
  public static void main(String[] args){
    CubbyHole c    = new CubbyHole();
    Producer  prod = new Producer(c);
    Consumer  cons = new Consumer(c);
    prod.start();
    cons.start();}
}
```

Fig. 17. JAVA : An example program – part II

6.2 Translation

In this section we shall explain those parts of the translation that are related to the `wait()` and `notify()` methods. The full translation can be found in appendix A. The full translation has been verified correct with respect to the assertion using the SPIN modelchecker. Modifying the assertion to a "*wrong*" statement in fact does result in error traces. For example, modifying the assertion in the JAVA program for each i to (modifying the right hand side of '`==`'):

$$\text{Verify.assert("received ok",received[i]} == \text{i+1);}$$

does in fact result in an error trace when translating this.

Figure 18 shows the definition of the `CubbyHole_Class` record type. In addition to the `LOCK` field (and the user defined fields) there are two fields administrating waiting processes (those that have called `wait` and which have not yet been notified). The field `WAIT` is a channel, and a process basically executes a `wait` by executing `WAIT?continue` where `continue` is the value `0` (just some signal). Hence, the waiting process will hang until the `continue` signal is sent by some other process executing a `notify` or `notifyAll`. The `WAITING` field is used to count the number of waiting processes, and is used by `notofyAll` to actually *free* them all by sending just as many `continue` signals on the `WAIT` channel.

```
typedef CubbyHole_Class{
  int  LOCK;
  int  WAITING;
  chan WAIT = [0] of {bit};
  int  contents;
  bool available;
};
```

Fig. 18. SPIN : Introducing waiting counter and channel

The macros `wait`, `notify` and `notifyAll` are shown in figure 19. The macros for accessing the `WAIT` and `WAITING` fields all refer directly to the `CubbyHole_Obj` array, which is the array of `CubbyHole_Class` records. In case there are several classes with synchronized methods, these macros must be made conditioned on `obj.class` to access the correct object arrays.

Note how the `wait` macro unlocks the object (to let other processes get access) before it actually "hooks up" on the `WAIT` channel of the object. The `notify` macro only sends a `continue` signal if there are processes waiting, thereby avoiding the notifying process to hang in case there are none waiting. The `notifyAll` macro repeatedly sends the `continue` signal as many times as there are waiting processes.

Finally, figure 20 shows the `code` macros for the `get` and `put` methods in the `CubbyHole` class. Note that these macros are then called within the `CubbyHole_get` and `CubbyHole_put` macros that do the locking.

The `CubbyHole_get_code` macro is parameterized with a variable that will be updated with the result value (recall that the `get` method in the JAVA program returns the value it consumes). The macro waits until a value is available. The `CubbyHole_put_code` macro on the other hand is parameterized with the value to be produced. It waits until no value is available, and a new can be made so.

```
#define continue 0

#define get_WAITING(obj)
  CubbyHole_Obj[obj.index].WAITING

#define set_WAITING(obj,value)
  CubbyHole_Obj[obj.index].WAITING = value

#define get_WAIT(obj)
  CubbyHole_Obj[obj.index].WAIT

#define wait(obj)
  atomic{
    unlock(obj);
    set_WAITING(obj,get_WAITING(obj) + 1);
    get_WAIT(obj)?continue;
    lock(obj)}

#define notify(obj)
  atomic{
    if
    :: get_WAITING(obj) > 0 ->
         get_WAIT(obj)!continue;
         set_WAITING(obj,get_WAITING(obj) - 1)
    :: else -> skip
    fi}

#define notifyAll(obj)
  atomic{
    do
    :: get_WAITING(obj) > 0 ->
         get_WAIT(obj)!continue;
         set_WAITING(obj,get_WAITING(obj) - 1)
    :: else -> break
    od}
```

Fig. 19. SPIN : waiting and notification

```
#define CubbyHole_get_code(obj,return_value)
  do
  :: CubbyHole_get_available(obj) == false -> wait(obj)
  :: else -> break
  od;
  CubbyHole_set_available(obj,false);
  notify(obj);
  return_value = CubbyHole_get_contents(obj)

#define CubbyHole_put_code(obj,value)
  do
  :: CubbyHole_get_available(obj) == true -> wait(obj)
  :: else -> break
  od;
  CubbyHole_set_contents(obj,value);
  CubbyHole_set_available(obj,true);
  notify(obj)
```

Fig. 20. Spin : Translation of class `CubbyHole` – relevant parts

7 Suspend and Resume

The `suspend()` method of the `Thread` class suspends a running thread. The `resume()` method resumes it from where it left off. If the thread that is suspended is inside (has called) a synchronized method, and hence has locked the object the method was called on, then it will maintain this lock during its suspension.

7.1 Example JAVA Program to be Translated

We shall design an example to illustrate in particular these methods. Before we proceed, we shall however extend the class `Verify` with a couple more specification methods.

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

  public static boolean eventually(boolean b){return true};

  public static void write(String s, boolean b){
    System.out.println(s + b);
  };
};
```

Fig. 21. JAVA : Extending the "logic"

7.1.1 More about Specifying Properties The new specification methods are shown in figure 21. The method `eventually` is like `assert` except that it is supposed to state a property which is specified to hold *eventually* – in the future. Hence, if one in the main program (the `main` method) writes:

```
Verify.eventually(x == 0)
```

then this states that in all traces, eventually `x == 0`, just as if we had written "`<> (x==0)`" in SPIN. The body of this method is "`return true`" since JAVA requires a function to contain a `return` statement of some kind. Basically, a call of the method just represents an information to the translator to generate an LTL formula to be verified in SPIN. Note how we keep within the JAVA language for writing specifications.

The method `write` simply writes the contents of a variable, boolean in this case. There should possibly be many such – for different types of values. The idea here is that calls of this method can be translated into print statements in SPIN, which print on the Message Sequence Charts during simulations, for example, simulations of error traces. This gives very readable graphical output which is easy to follow.

7.1.2 The JAVA Program

Now to the JAVA program itself, see figure 22. It shows two how two threads, `a1` and `a2` of types `Agent1` and `Agent2`, are spawned, which both update the "global" variables defined in the class `Status`. Note that these variables are all `static`, meaning that they are associated with the class, and not with objects of that class. That is, there is only one copy of these values, common for all objects. One can refer to the static variables by just prefixing with the class name as in "`Status.a`", without creating objects.

`Agent1` basically assigns `true` to `Status.a`, and then suspends itself. `Agent2` assigns `true` to `Status.b` and then resumes `Agent1`. From the `main` method it can be seen how the `Agent2` thread is initialized with the `Agent1` thread (`a1`) as parameter. When `Agent1` has been resumed, it finally updates `Status.c`.

The `main` method contains a call of `Verify.eventually` and a call of `Verify.assert`. These properties state, respectively, that "*eventually* `Status.c` *will become true*" (meaning that `Agent1` terminates), and "*whenever* `Status.c` *is true, so is* `Status.a` *and* `Status.b`" (meaning for example that `Agent2` has executed – to resume `Agent1`).

```
class Status{
  public static boolean a = false;
  public static boolean b = false;
  public static boolean c = false;
};

class Agent1 extends Thread{
  public void run(){
    Status.a = true;
    Verify.write("a == ",Status.a);
    suspend();
    Status.c = true;
    Verify.write("c == ",Status.c);}
};

class Agent2 extends Thread{
  Thread other;

  public Agent2(Thread other){
    this.other = other;};

  public void run(){
    Status.b = true;
    Verify.write("b == ",Status.b);
    other.resume();}
};

class Main{
  public static void main(String[] args){
    Verify.eventually(Status.c);
    Agent1 a1  = new Agent1();
    Agent2 a2  = new Agent2(a1);
    a1.start();
    a2.start();
    Verify.assert("[](c => a & b)",
      !Status.c || (Status.a & Status.b));};
}
```

Fig. 22. Java : An example program

7.1.3 More About the Specification Note that the call of `Verify.assert` occurs after the start of all threads. This position of the call in fact turns it into an invariant to be checked since it may get executed at any point in time during the whole programs execution. Hence, general invariant properties may be formulated as assertions written at the end of the `main` method. The call of `Verify.eventually` is also strategically positioned before any other statement, simply to make it cover all traces from their beginning. This will be discussed in some more detail in section 14.

When executing this program we will see the output:

```
a == true
b == true
c == true
```

Generally, on a typical JAVA platform this will be the output: first `Agent1` will execute, and suspend itself; then `Agent2` will execute, and and resume `Agent1`; then `Agent1` will continue, and terminate; (and then the assertion in the `main` method will be executed). That is, many JAVA platforms do not interleave process execution unless forced to by explicit statements like for example `yield`. Since the `Agent1` thread is started first, it will on our Sun stations execute first until it suspends itself. This is not the case on all platforms. To quote [2]:

> The Java runtime environment does not implement (and therefore does not guarantee) time–slicing. However, some systems on which you can run Java programs do support time–slicing. Your Java programs should not rely on time-slicing as it may produce different results on many systems.

This in fact means that a JAVA program cannot be tested fully on for example a non-time-sliced platform. In the above case, all tests will show that the `Status.c` variable eventually becomes true. This is, however, not the case on a time-sliced platform. The reader may try to guess why. In section 7.2.5 it will be explained how SPIN disproves this eventually–property for the translated version, giving an error trace demonstrating what goes wrong. The point here is, that the SPIN translation *will!* model full time-slicing.

7.2 Translation

7.2.1 General Ideas A SPIN `proctype` declaration can be suffixed by an optional "`provided (bool_exp)`" clause to constrain its execution to those global system states for which the corresponding expression (the expression can contain global references, or references to the process's _pid) evaluates to true. We shall use this construct to model suspension and resumption. We introduce an array which maps each process identifier to a boolean, being true if that process has been suspended. The provided clause for a process is then the falsity of this array applied to the corresponding process id. This translation has a serious drawback in that it prevents the use of SPIN's partial order reduction algorithm, but at the current moment we don't have a better suggestion. If a program contains no thread suspension one does of course not need to introduce these `provided` clauses and the partial order reduction *can* in this (hopefully most typical) case be applied.

Before we proceed with the more important part of the translation, let us introduce the standard types and constants, see figure 23. The `failure` will be used as the last *else* branch in conditional statements. The `write` macro writes the contents of a variable. Note that the `x` in the text string also gets replaced at macro expansion time.

```
type Index = byte;
type ClassName = Status,Agent1,Agent2,Main;
typedef ObjRef{ClassName class; Index index};
type ProcId = byte

#define false 0
#define true  1
#define undefined 0
#define MAX 5
#define this _pid
#define failure else -> assert(false)

#define write(x)
  printf("MSC: x == %u",x)
```

Fig. 23. SPIN : Some types etc.

Figures 26 and 27 show the translation of the classes `Agent1` and `Agent2`, and in particular the definition of the typedefs `Agent1_Class` and `Agent2_Class`, which both include a field named `PID`, representing the process identification (generated by SPIN) of the process associated with an object. The macros for suspension and resumption of processes in figure 24 access this field. The `suspended` array maps process identifiers to booleans (`true`, if suspended). The macro `set_PID` assigns a value to the `PID` field. The macro `running` will be used as the boolean expression in the `provided` clauses. The macros `suspend` and `resume` basically just assigns, respectively, `true` and `false` to the relevant process ids in the `suspended` array. these process ids are looked up in the data area associated to the object being suspended/resumed. The `start` method is used to start threads atomically such that the process id can be stored safely in time.

```
ProcId suspended[MAX];

#define set_PID(obj,value)
  if
  :: obj.class == Agent1 -> Agent1_Obj[obj.index].PID = value
  :: obj.class == Agent2 -> Agent2_Obj[obj.index].PID = value
  fi

#define running
  !suspended[this]

#define suspend(obj)
  if
  :: obj.class == Agent1 ->
       suspended[Agent1_Obj[obj.index].PID] = true
  :: obj.class == Agent2 ->
       suspended[Agent2_Obj[obj.index].PID] = true
  :: failure
  fi

#define resume(obj)
  if
  :: obj.class == Agent1 ->
       suspended[Agent1_Obj[obj.index].PID] = false
  :: obj.class == Agent2 ->
       suspended[Agent2_Obj[obj.index].PID] = false
  :: failure
  fi

#define start(thread,obj)
  atomic{
    pid = run thread(obj.class,obj.index);
    set_PID(obj,pid)}
```

Fig. 24. Spin : Suspending and resuming operations

7.2.2 Translation of Class `Status` Note that all variables in the `Status` class are static, which means that they occur in one copy each, rather than being copied for each new object. This means that we only need to define one data area as done in figure 25.

7.2.3 Translation of Class `Agent1` The translation of class `Agent1`, see figure 26 follows the standard pattern, except for the `provided` clause. The translation of this example did in fact lead to the discovery of a bug in Spin (has been corrected by Gerard Holzmann in the new soon coming version) where an

```
typedef Status_Static{
  bool a = false;
  bool b = false;
  bool c = false;
};
Status_Static Status_Static_Area;
```

Fig. 25. SPIN : Translation of class `Status`

object of type `ObjRef` could not be passed as a parameter to a process. Therefore, in this example, we transfer its components `class` and `index`, and immediately combine them into `obj`, the object which the "thread" is part of. Hence, when the process executes `suspend(obj)`, it suspends *itself*.

```
typedef Agent1_Class{ ProcId  PID }; Agent1_Class Agent1_Obj[MAX];
  Index Agent1_Next = 0;

#define Agent1_constr(obj)
  ObjRef obj;
  obj.class = Agent1;
  atomic{obj.index = Agent1_Next; Agent1_Next++}

proctype Agent1_Thread(ClassName class; Index index)
  provided (running)
{ ObjRef obj;
  obj.class = class;
  obj.index = index;
  Status_Static_Area.a = true;
  write(Status_Static_Area.a);
  suspend(obj);
  Status_Static_Area.c = true;
  write(Status_Static_Area.c);
};
```

Fig. 26. SPIN : Translation of class `Agent1`

7.2.4 Translation of Class `Agent2`

The translation of class `Agent2`, see figure 27 follows the same pattern, except now the thread suspends "*the other*" thread, the `Agent1` thread stored in the variable `other` given as parameter to the constructor.

```
typedef Agent2_Class{
  ProcId PID;
  ObjRef   other
};
Agent2_Class
Agent2_Obj[MAX]; Index Agent2_Next = 0;

#define Agent2_get_other(obj)
  Agent2_Obj[obj.index].other

#define Agent2_set_other(obj,value)
  Agent2_Obj[obj.index].other.class = value.class;
  Agent2_Obj[obj.index].other.index = value.index

#define Agent2_constr(obj,other)
  ObjRef obj;
  obj.class = Agent2;
  atomic{obj.index = Agent2_Next; Agent2_Next++};
  Agent2_set_other(obj,other)

proctype Agent2_Thread(ClassName class; Index index)
  provided (running)
{ ObjRef obj;
  obj.class = class;
  obj.index = index;
  Status_Static_Area.b = true;
  write(Status_Static_Area.b);
  resume(Agent2_get_other(obj));
};
```

Fig. 27. SPIN : Translation of class `Agent2`

7.2.5 Translation of Class `Main` The `Main` method constructs the two thread objects and starts them, see figure 28. Since the `assert` statement occurs at the end, it will represent an invariant property to be verified, since its execution may occur at *any time*! after the start of the processes. Hence, it must be true at any time.

The eventuality property of the JAVA program (that `Status.c` eventually becomes true) must be formulated as an LTL formula of the form "`<>Status_Static_Area.c`" (using a macro name though for the expression following "`<>`" as required by SPIN). As it turns out, the assertion is true (whenever `c` is true also `a` and `b` are true), while the LTL eventually property is not true: in some traces the variable `c` is never set to true. SPIN returns an error trace where the `Agent2` thread `a2` executes the `resume` method to resume the `Agent1` thread `a1` *before!* this suspends itself.

```
init{
  ProcId pid;
  Agent1_constr(a1);
  Agent2_constr(a2,a1);
  start(Agent1_Thread,a1);
  start(Agent2_Thread,a2);

  assert(!Status_Static_Area.c ||
         (Status_Static_Area.a & Status_Static_Area.b));
}
```

Fig. 28. SPIN : Translation of class `Main`

7.3 Alternative Translations

In the translation above an array (`suspended`) maps a process identifier to the boolean value `true` in case that process has been suspended. An alternative is to use an array to model a list of all suspended processes, and then look through the list for each process execution:

```
member(_pid, suspended)
```

where `member` performs a scan of the list. Alternatively, the list may be implemented as a channel, and `member` may then simply be implemented as:

```
suspended??[eval(_pid)]
```

being true if `_pid` is in the channel.

Using an array as done in the presented translation appears to be the most time efficient solution. One must remember that this check has to be performed for each execution of a process being associated with a `provided` clause. The drawback of the array solution is that one needs an entry in the array for each possible process identifier in the program. This may require more space than the *list* solutions, where one just needs to store exactly the suspended processes.

8 Stop, Join, isAlive

The `stop` method in the `Thread` class kills a thread by throwing a `ThreadDeath` exception inside the thread that is caught at the outer level of the thread, causing its death, unless the exception is caught by the user defined application program inside the thread. The `isAlive` method returns true if the thread has started and has not been stopped. Note that `isAlive` hence is not true before `start` has been called on that same object. The `join` method returns if the object is not alive. That is, when `isAlive` is false. Hence, a `join` executed on a thread that has not yet been started will return because `isAlive` is false.

When a thread is stopped it may be in the middle of executing synchronized methods on several different objects. Each of these objects are locked to prevent other threads from executing synchronized methods in parallel. The locks on these objects have to be released as a result of the call of `stop`.

8.1 Example Java Program to be Translated

8.1.1 The Java program the Java program to be translated is shown in figure 29. The class `Obj` contains a collection of static variables, hence global to the program. The first three (`data`, `job` and `terminator`) will be assigned object values, the objects of the program. Two of these objects (`job` and `terminator`) are the treads of the program. This style of defining all objects globally as static variables is an alternative to creating them inside the `main` method and then passing them around as parameters as we have seen in previous examples (see for example figures 16 and 17).

The `Data` class (of which `Obj.data` will be an instance) contains two synchronized methods `work` and `finalize`. The `work` method performs an infinite loop, and hence will never terminate unless the thread that calls `work` is stopped violently from outside, for example by a call of `stop`. Now, the `Job` thread does in fact call `work` while the `Stop` thread stops the `Job` thread (`Obj.job`). The `main` method starts the `Job` and `Stop` threads `Obj.job` and `Obj.terminator`, and then waits to `join` the `Obj.job` thread. The join succeeds when the `Obj.terminator` thread stops the `Obj.job` thread. The `finalize` method is called at the end to demonstrate that in fact the lock put on the `Obj.data` object by the `Obj.job` thread has been released.

```java
class Obj{
  public static Data data;
  public static Job  job;
  public static Stop terminator;
  public static boolean finalized = false;
};

class Data{
  public synchronized void work(){
    while (true) {/* do some job*/};
  };

  public synchronized void finalize(){
    Obj.finalized = true;
  };
}

class Job extends Thread{
  public void run(){
    Obj.data.work();
  };
};

class Stop extends Thread{
  public void run(){
    Obj.job.stop();
    Verify.eventually(Obj.finalized);
  };
};

class Main{
  public static void main(String[] args){
    Obj.data = new Data();
    Obj.job = new Job();
    Obj.terminator = new Stop();
    Verify.eventually(Obj.finalized);
    Obj.job.start();
    Obj.terminator.start();
    try{Obj.job.join();}catch(InterruptedException e){};
    Verify.assert("job has stopped",!Obj.job.isAlive());
    Obj.data.finalize();
  };
};
```

Fig. 29. Java : An example program

8.1.2 The Specification

The program contains three properties we want to prove, two of which occur in the `main` method and one of which occur in the `run` method of the `Stop` class. The property:

```
Verify.eventually(Obj.finalized)
```

in the `main` method states that the main program will always reach its end and call the `finalize` method. This requires that the `join` call always succeeds and terminates. The `assert` statement in the `main` method states that after the join, the `Obj.job` thread is no longer alive. The third property:

```
Verify.eventually(Obj.finalized)
```

in the `run` method of the `Stop` class states that once the `Obj.job` thread had been stopped, eventually the `main` method will join, and the `finalize` method will be called, updating `Obj.finalized` to true. This is an example of a LTL property *embedded* "deep" inside a program, and hence not at the beginning of the program (as the first thing in the `main` method as the other eventually-property above). The semantics is therefore different in the sense that this property should only hold "*from this point on*". That is, only when we have really stopped the `Obj.job` thread we can be sure that the `join` in the `main` method later will be executed. As we shall see, only the latter embedded LTL property and the `assert` property hold, whereas the first eventually-property in the `main` method does not hold. That is, not in all execution traces will `finalize` eventually be called.

8.2 Translation

8.2.1 General Ideas

The translation is based on introducing a new boolean `STOP` field in the data area of a thread object, initially `false`, and wrapping an `unless` statement around the run-code of the thread, which exits as soon as the `STOP` field gets the value `true`. The `STOP` field is assigned the value `true` by the `stop` method. Note that the `stop` method in JAVA in fact throws a `ThreadDeath` exception. Our translation will not reflect this fact, since we are not yet decided upon a satisfactory translation of exceptions. Once we know how to translate exceptions correctly, the translation of `stop` must be changed accordingly.

In a addition to the `STOP` field, a `PID` field will contain the process identification of the object (if it is a thread) as demonstrated in section 7. This field is initially `null`. Hence `isAlive` is true whenever `PID` is different from `null` (the thread has been started) and `STOP` is false.

Recall further that in order to model synchronized methods we also introduce a `LOCK` field in the data area of those objects having synchronized methods. This field will point to the thread currently locking the object. When a thread is stopped, all objects that it has locked in this way must be released. For this purpose a list valued variable (a channel in fact) named LOCKING is furthermore introduced in each thread (proctype), which contains references to those objects the thread has locked. When the thread is stopped abnormally this list is emptied, and each of its objects is unlocked.

Figure 30 introduces the standard collection of types and constants. The only new thing is the definition of the macro list as a different name for chan. It will be used for defining "variables" intended to hold lists of elements. This technique was also used in [3].

```
type Index = byte;
type ClassName  =  {Obj,Data,Job,Stop,Main};
typedef ObjRef{ClassName class; Index index};
type ProcId = int

#define false 0
#define true  1
#define undefined 0
#define MAX 5
#define null -1
#define this _pid
#define failure else -> assert(false)
#define list chan
```

Fig. 30. SPIN : Some types etc.

Figure 31 shows the definitions of macros needed for locking and unlocking objects having synchronized methods, in this case only objects of the class Data. The macros get_LOCK and set_LOCK just reads and writes to the LOCK field as we have seen before.

The macros add_LOCKING, remove_LOCKING and clear_LOCKING are used to guarantee that locked objects get unlocked in case of a violent stop. The macro add_LOCKING adds an object being locked to the list valued LOCKING variable local to each thread proctype as wee shall see, and remove_LOCKING removes this lock after the synchronized method terminates normally. In case of an abnormal stop, the macro clear_LOCKING goes though all the locked objects and unlocks them, one by one.

The macros lock and unlock locks and unlocks a single object in the normal situation. Each of these operations requires an update of the LOCK field in in object and of the LOCKING field in the calling thread.

Figure 32 shows the macros needed for stopping and joining threads. The first four macros just access the fields PID and STOP. The definitions of isAlive, join, and stop are self-explanatory, while the abort macro needs a bit of explanation.

```
#define get_LOCK(obj)
  Data_Obj[obj.index].LOCK

#define set_LOCK(obj,value)
  Data_Obj[obj.index].LOCK = value

#define add_LOCKING(obj)
  LOCKING!obj

#define remove_LOCKING(obj)
  LOCKING??(eval(obj.class),eval(obj.index))

#define clear_LOCKING
  ObjRef locked_obj;
  do
  :: LOCKING?(locked_obj) ->
       set_LOCK(locked_obj,null)
  :: empty(LOCKING) -> break
  od

#define lock(obj)
  atomic{
    get_LOCK(obj) == null ->
      set_LOCK(obj,this);
      add_LOCKING(obj)}

#define unlock(obj)
  atomic{
    set_LOCK(obj,null);
    remove_LOCKING(obj)}
```

Fig. 31. SPIN : Synchronization locking

This macro is called in a thread when it becomes stopped and constitutes the exit-condition in the `unless` construct that is wrapped around the run–code. Hence, it gets executed as soon as the guard `get_STOP(obj)` evaluates to true. Thereafter it clears all locks owned by the thread.

```
#define set_PID(obj,value)
  if
  :: obj.class == Job  -> Job_Obj[obj.index].PID = value
  :: obj.class == Stop -> Stop_Obj[obj.index].PID = value
  :: failure
  fi

#define get_PID(obj)
  (obj.class == Job -> Job_Obj[obj.index].PID :
    (obj.class == Stop -> Stop_Obj[obj.index].PID : undefined))

#define set_STOP(obj,value)
  if
  :: obj.class == Job  -> Job_Obj[obj.index].STOP = value
  :: obj.class == Stop -> Stop_Obj[obj.index].STOP = value
  :: failure
  fi

#define get_STOP(obj)
  (obj.class == Job -> Job_Obj[obj.index].STOP :
    (obj.class == Stop -> Stop_Obj[obj.index].STOP : undefined))

#define isAlive(obj)
  (get_PID(obj) != null & get_STOP(obj) != true)

#define join(obj)
  !isAlive(obj)

#define stop(obj)
  set_STOP(obj,true)

#define abort(obj)
  atomic{get_STOP(obj); clear_LOCKING}

#define start(thread,obj)
  atomic{
    pid = run thread(obj.class,obj.index);
    set_PID(obj,pid)}
```

Fig. 32. Spin : stop, isAlive and join

8.2.2 Translation of Class `Obj`

All variables in the `Obj` class are static, which means that they occur in one copy each, rather than being copied for each new object. We model this differently than in figure 25 due to an error in the SPIN model checker (the type checker protests when a nested record is sent over a channel). Hence, in figure 33 we instead introduce a variable for each field in the class.

```
ObjRef Obj_Static_data;
ObjRef Obj_Static_job;
ObjRef Obj_Static_terminator;
bool   Obj_Static_finalized = false;
```

Fig. 33. SPIN : Translation of class `Obj`

8.2.3 Translation of Class `Data`

The translation of this class follows the same pattern as in section 5 where each method `M` is mapped into two macros: `C_M_code` and `C_M` where `C` is the class name. Observe that the constructor is called `Data_new` instead of `Data_constr`. The difference between this an earlier ones is that it does not contain the declaration "`ObjRef obj`" since it is supposed to be called in a context where the object variable has already been declared. It is the difference between the JAVA statements:

```
Data data = new Data()
```

being mapped into a call of `Data_constr(data)` (but it does no occur in this program) and

```
Data data;
...
data = new Data();
```

being mapped into a separate declaration of `data` and then a call of `Data_new(data)`.

```
typedef Data_Class{
  ProcId LOCK = null
};
Data_Class Data_Obj[MAX];
Index Data_Next = 0;

#define Data_new(obj)
  obj.class = Data;
  atomic{obj.index = Data_Next; Data_Next++}

#define Data_work_code(obj)
  do
  :: skip
  od

#define Data_work(obj)
  if
  :: get_LOCK(obj) == this ->
       Data_work_code(obj)
  :: else ->
       lock(obj);
       Data_work_code(obj);
       unlock(obj)
  fi

#define Data_finalize_code(obj)
  Obj_Static_finalized = true

#define Data_finalize(obj)
  if
  :: get_LOCK(obj) == this ->
       Data_finalize_code(obj)
  :: else ->
       lock(obj);
       Data_finalize_code(obj);
       unlock(obj)
  fi
```

Fig. 34. Spin : Translation of class Data

8.2.4 Translation of Class Job

In the translation of class Job, figure 35, we see the introduction of the fields PID and STOP. The Job_Thread proctype has two parameters due to the bug discovered in SPIN and as explained in section 7.2.3. The variable (channel) LOCKING is introduced here to contain references to all locked objects, to be released upon abnormal termination (stop). Note how the run-code is surrounded by an unless construct conditioned by the abort(obj) call. That is, as soon as the abort code gets executable the call of Data_work will terminate, and the abort code gets executed. In case of normal termination the STOP field must be set to true as the last action.

```
typedef Job_Class{
  ProcId PID = null;
  bool STOP = false
};
Job_Class Job_Obj[MAX];
Index Job_Next = 0;

#define Job_new(obj)
  obj.class = Job;
  atomicobj.index = Job_Next; Job_Next++

proctype Job_Thread(ClassName class; Index index){
  ObjRef obj;
  list LOCKING = [MAX] of {ObjRef};
  obj.class = class;
  obj.index = index;
  {Data_work(Obj_Static_data)}
    unless
  {abort(obj)};
  set_STOP(obj,true);
};
```

Fig. 35. SPIN : Translation of class Job

8.2.5 Translation of Class Stop

The translation of class Stop is shown in figure 36. The new thing to observe here is the declaration of the variable JOB_STOPPED (initially false), and the assignment of true to it at the position where the eventually property:

```
Verify.eventually(Obj.finalized)
```

occurs in the run method of the JAVA class Stop. The eventually-property is then translated into:

```
[](JOB_STOPPED => <>Obj_Static_finalized)
```

reading as follows: "*it is always the case that whenever the variable* `JOB_STOPPED` *becomes true, then eventually the variable* `Obj_Static_finalized` *becomes true*". In order for this property to hold, it must hold for all traces (this is the semantics of LTL), and it must hold from the *beginning* (first state) of each of these traces – and it does. This is a general way to translate embedded LTL in JAVA. For each such formula one introduces a variable that becomes true at that point, and then the translated formula is guarded with this variable. In a fully general translation, one needs such a variable for each object, since the property must be true for all objects. Hence, such boolean flags must be part of the data area associated with objects.

```
typedef Stop_Class{
  ProcId PID = null;
  bool STOP = false
};
Stop_Class Stop_Obj[MAX];
Index Stop_Next = 0;

#define Stop_new(obj)
  obj.class = Stop;
  atomic{obj.index = Stop_Next; Stop_Next++}

bool JOB_STOPPED = false;

proctype Stop_Thread(ClassName class; Index index){
  ObjRef obj;
  list LOCKING = [MAX] of {ObjRef};
  obj.class = class;
  obj.index = index;
  {
   stop(Obj_Static_job);
   JOB_STOPPED = true;
  }
    unless
  {abort(obj)};
  set_STOP(obj,true);
};
```

Fig. 36. SPIN : Translation of class `Stop`

8.2.6 Translation of Class `Main`

The `main` method in the JAVA program constructs the three objects `Obj.data`, `Obj.job` and `Obj.terminator`. This translates into calls of the macros Data_new, Job_new and Stop_new which we recall do not declare these objects as they have already been declared (as static variables in the class Obj). Note that in the JAVA program the call of join is protected by a try ... catch construct. This is not translated here as we have not yet found a fully satisfactory translation of exceptions.

```
init{
  ProcId pid;
  list LOCKING = [MAX] of {ObjRef};
  Data_new(Obj_Static_data);
  Job_new (Obj_Static_job);
  Stop_new(Obj_Static_terminator);
  start(Job_Thread ,Obj_Static_job);
  start(Stop_Thread,Obj_Static_terminator);
  join(Obj_Static_job);
  assert(!isAlive(Obj_Static_job));
  Data_finalize(Obj_Static_data);
}
```

Fig. 37. SPIN : Translation of class Main

When applying the SPIN modelchecker, the assert statement is verified to hold. The eventually-property occurring in the main method of the JAVA program, figure 29 is translated into the LTL following formula (ignoring here that all propositions have to be macro calls):

```
<> Obj_Static_finalized
```

That is, for all traces of the program, eventually the variable Obj_Static_finalized becomes true. SPIN rejects the property as being true and returns an error trace illustrating a situation where the terminator never kicks in and stops the job thread. Hence, the job thread loops forever. This in fact is also the effect of running the JAVA program on our platform since the terminator thread is started after the job thread, meaning that it will never get a time slice.

9 Yield and Sleep

The `yield()` method stops executing a thread and returns control to the scheduler. So if there is another runnable task at the same or higher priority, it will be run. This is not quite a no-op in SPIN, because a no-op would allow the same thread to continue running, whereas `yield()` enforces a switch. So an error trace where the same thread kept running after calling `yield()` might not actually occur in the running of the actual JAVA program; this would be a false negative.

The `sleep(int millisecs)` method suspends the thread for a given amount of time, at which point it becomes runnable again. We haven't thought significantly about modeling real-time in SPIN, so `sleep` and `yield` would have the same translation.

10 Data Structures

JAVA has vectors, etc. We haven't thought much about them; they can perhaps be modeled using channels as in the RA Report.

11 Expressions

A statement such as

```
a = a + 1
```

is atomic in SPIN but not in JAVA. This can be modeled to some extent by a transformation that introduces temporary variables, e.g. For example, the JAVA statement:

```
a = b * c + d
```

can be translated into (using temporary variables `t1` ... `t5`):

```
t1 = b;
t2 = c;
t3 = t1 * t2;
t4 = d;
t5 = t3 + t4;
a = t2
```

We are not sure whether the order of evaluation of operations like `+` is predefined in JAVA. So that means that whichever order we decide to introduce, temporaries may not model a particular JAVA compiler. So it is conceivable that some interleaving allowed by JAVA won't be possible in the translation, and that only that interleaving would cause a bug.

A JAVA expression tree could instead be translated to a data flow model, where each node in the tree is represented by a process that accepts inputs from children nodes and passes a value to its parent. Although it would capture all possible interleavings, it would be expensive.

12 Exceptions

A preliminary thought is to have a global variable that holds the exception object that has been raised. Translate "`try...catch`" into an "`unless`" that tests the type of exception to see if it should catch it.

A major issue here is that in case of nested `unless` statements in SPIN, the outermost "unless" will fire. This is the opposite of JAVA where the "lowest" applicable "try" captures an exception.

13 Priorities

The semantics of priorities in JAVA seems to be as follows. There is assumed to be a scheduler of threads. Whenever the scheduler has a choice of which thread to run, it chooses a highest priority thread. A thread's priority can be set using the `setpriority` method of the `Thread` class.

The priority of a thread could be modeled as another field of the `typedef` representing the thread. But how can the scheduler be modeled? A solution is to intersperse between every pair of statements a statement that blocks if there is a higher-priority thread that can be run. However, this solution is somewhat involved.

As the Mars pathfinder bug demonstrated, real-time starvation can occur in systems of concurrent processes with priorities. By "real-time starvation" we mean that process A prevents process B from achieving its real-time deadlines because A's priority is higher. It is worth thinking about whether this sort of thing can be modeled in PROMELA.

We need a `provided` clause that blocks unless the priority of the process is the highest of the runnable processes. How do we determine the highest priority runnable process? We could iterate through the runnable thread objects, finding the maximum priority. Is it possible to tell if a process is runnable? One can check the `STOP` field of the corresponding thread object, but how do we determine if a process is, say, waiting on a synchronized method?

14 Specifications

In this section we shall elaborate on the logic for specifying the properties of Java programs. We have already used a certain style of specification in the previous Java programs, and in this section we shall outline the full logic together with an outline of its semantics in terms of its translation into Spin's Ltl. Our main suggestion is not to extend the Java language but to express temporal properties as calls to methods defined in a special temporal logic class, all of whose methods are static (hence one does not need to instantiate the class to objects before calling the methods).

The first section elaborates in the scheme already used in previous sections. The second section discusses a slightly different approach (although basically the same solution) where one can avoid prefixing temporal operators with class names by using inheritance instead. Then follows an idea based on specifying properties as declarations of methods instead of in terms of statements. The last section shortly discusses completely alternative solutions, one based on expressing specifications as comments and one simply by extending the syntax of Java. Neither of these two solutions are recommended for the moment. Even comments need a special parser since Java parsers just ignore the contents of comments.

14.1 The `Verify` Class

14.1.1 Syntax and Outline of Semantics Figure 38 shows the class `Verify` which provides a range of temporal operators. In fact, the expressive power of this "*language*" is the same as that of Spin's Ltl. Each method is defined as returning a boolean value, except the `assert` method. This allows nested formulae as can be seen in figure 39 where the `Verify` class is applied. The methods just return `true` (except `assert`) since there is no simple way to calculate their result within Java and since their only purpose hence is to inform the translator about which Ltl properties to generate for the Spin program resulting from the translation. Only calls of these methods will result in translation.

Associated with each method is a comment explaining how a call of this method is translated into Ltl. The translation is of course recursive in case of nested formulae. Some of the methods are just introduced to make life easier, like for example `response`. That is, the second formula in figure 39 can be written as "`Verify.response(A,B)`". Note that implication is not part of Java's operators and hence we must use the rule of logic: $(p \rightarrow q) = ((\neg p) \vee q)$. Alternatively, the method `implies` has been introduced.

14.1.2 More about Semantics The formulae in figure 39 all occur at the start of the Java programs `main` method. This means that they can be translated directly into Ltl formulae as indicated by the comments in figure 38. This

```
class Verify{
  public static void assert(String s, boolean  p){
    if (!p) System.out.println("assertion " + s + " broken");};
  /* p */

  public static boolean always(boolean p){return true;};
  /* []p */

  public static boolean eventually(boolean p){return true;};
  /* <>p */

  public static boolean until(boolean p, boolean q){return true;};
  /* p U q */

  public static boolean response(boolean p, boolean q){return true;};
  /* [](p -> <>q) */

  public static boolean precedence(boolean p, boolean q, boolean r)
    {return true;};
  /* [](p -> (q U r)) */

  public static boolean implies(boolean p, boolean q){return true;};
  /* p -> q */
};
```

Fig. 38. JAVA : The `Verify` class

is because an LTL formula in SPIN is true of a program if it is true from *the beginning*! (first state) of all traces of the program. These formulae are, however, also allowed to occur embedded inside JAVA code, and in that case their translation is a little bit more subtle. Generally, an embedded call of a `Verify` method translates into the assignment of `true` to an automatically generated system variable (initially `false`), and to the generation of an LTL formula which is guarded by the truth of that variable. Suppose for example the following piece of JAVA code "deeply" nested inside a JAVA program:

```
...
Verify.eventually(P);
...
```

This will then translate into a SPIN program of the form:

```
...
VERIFY_VARIABLE = true
...
```

```
class Main{
  public static void main(String[] args){
    boolean A = false;
    boolean B = false;

    Verify.eventually(A);
    Verify.always(!A || Verify.eventually(B));

    A = true;
    B = true;
  }
}
```

Fig. 39. Java : Using the `Verify` class

and then the Ltl formula (where `P'` is the translation of `P`):

```
[](VERIFY_VARIABLE -> <>P')
```

The translation is even more subtle than that. If the formula occurs inside a method of a class, then the formula must be true for all objects of that class. Hence, such a `VERIFY_VARIABLE` must be introduced in the data area for the objects. Furthermore, properties should be interpretated relative to the lifetime of the objects. That is for example, a property `Verify.always(P)` should be true if it is true during the lifetime of the object. That is, it should be translated into something like (note that the until operator `U` in Spin is strong meaning that its second argument must become true eventually, which need not be the case for `STOP`):

```
[](VERIFY_VARIABLE -> (([]P) or (P U STOP)))
```

That is, the property `P` must be true now and until the object stops, here ignoring the fact that the `VERIFY_VARIABLE` and the `STOP` variable are part of the data area of the object. A property of the form `Verify.eventually(P)` is translated into:

```
[](VERIFY_VARIABLE -> (!STOP U P))
```

That is, eventually `P` must become true, and in the meantime `STOP` must be false, meaning that the object is alive. Hence, `P` becomes true before the object dies. At least this must be the case for formulae that contains variables purely local to the object. If on the other hand the formula contains variables global to the object, or a mixture of global and local, then things get even more subtle. In fact, without restrictions the translation may become quite complicated, and perhaps not even feasible because SPIN's LTL formulae have to be defined statically, while object creation and death is dynamic. A lot to think about here.

Note by the way, that if an `assert` statement occurs at the end of the `main` method of the JAVA program it will function as a program invariant to be always true. This is because the SPIN model checker will do its best to falsify the assertion by executing it in all of the program's states. Recall that the `main` method becomes a process running in parallel with all the other spawned processes.

14.1.3 Correspondence to LTL We mentioned that the logic presented here is as expressive as SPIN's LTL. We can demonstrate this by showing how LTL formulae are translated into our logic. This is shown in table 40.

Nr.	LTL	JAVA Logic
1	`[]p`	`always(p')`
2	`<>p`	`eventually(p')`
3	`p U q`	`until(p',q')`
4	`p -> q`	`!p' || q'` or `implies(p',q')`
5	`p and q`	`p' & q'`
6	`p or q`	`p' || q'`
7	`not p`	`!p'`

Fig. 40. JAVA : Translating LTL into JAVA logic

The propositional LTL formulae 5 – 7 are translated as indicated in the table unless they occur at the outermost level. Recall that only calls of `Verify` methods will be translated into LTL. Hence, if propositional operators occur at the outermost level they must either be embedded in an `assert` statement, or one has to introduce `verify` methods for `and`, `or` and `not`. For example, the following three groups of formulae would state the same thing (`p` holds now and `q` holds eventually):

```
Verify.and(p,eventually(q));

Verify.assert(p);
Verify.eventually(q)

Verify.assert(p & eventually(q))
```

There is room for language debates here.

14.2 Interpretation of Verifications

An obvious question is how to interpretate verification results. That is, what does it imply that a formula is verified to hold, respectively not hold in the translated PROMELA program? We shall state an informal theorem answering these questions. We shall (here informally) assume that LTL (Linear Temporal Logic) formulae can be used to state properties about programs in both languages, assuming a satisfaction relation "$\models$" between programs and formulae (one for each language). Hence, $p \models \psi$ means that program p satisfies formula ψ.

Theorem 1 Interpretating verification results. *Assume a* JAVA *program J and its* PROMELA *translation P.*

Then the following holds for any LTL *formula ψ:*

$$P \models \psi \text{ implies } \mathrm{J} \models \psi \tag{1}$$

The following does however **not** *hold:*

$$\neg(P \models \psi) \text{ implies } \neg(\mathrm{J} \models \psi) \tag{2}$$

Informal Proof:

Proof of 1: The PROMELA *program P allows the same or more interleaving than the* JAVA *program J. Hence, the program J denotes a set of traces being a subset (in some interpretated way) of the set of traces denoted by P. Furthermore, an* LTL *formula is true for P if it is true for all traces of that P. Hence it will also be true for the traces of J. Proof of* (2)*: Consider a property ψ that does not hold for P because it does not hold for a certain single trace τ. Since the set of traces denoted by P may be a strict superset of the set of traces denoted by J, the latter may not include τ.*

Statement (1) of the theorem basically states that if a property is proved to hold for the translated PROMELA program then it also holds in the source JAVA program. That statement (2) does not hold means that an error trace generated on the basis of a verification does not necessarily mean an error in the JAVA program. This is because certain JAVA implementations do not support implicit time-slicing. Put differently, the following does **not** hold: "$J \models \psi$ implies $P \models \psi$", meaning that the JAVA program may have certain properties that the PROMELA program does not. In general, however, if a property does not hold in the PROMELA program it possibly represents a problem in the JAVA program since programs should not rely on lack of time-slicing in the platform they run upon.

14.3 Using Inheritance to Avoid Class Prefixes

If it becomes a goal to shorten temporal specifications, for example by avoiding writing the `Verify` prefix to all calls of methods inside `Verify` then a solution is to *inherit* these methods. There are basically two kinds of objects in JAVA (seen from the current perspective): non-threads and threads. Hence, we define the `Verify` class as before, but in addition we also define an *extension* of the `Thread` class introducing exactly the same methods. This is illustrated in figure 41. Now we can inherit these two classes, `Verify` for the non-thread classes and `VThread` for the thread classes, as shown in figure 42.

```
class Verify{
  // ... as before
  public static boolean eventually(boolean p){return true;};
};

class VThread extends Thread{
  // ... as before
  public static boolean eventually(boolean p){return true;};
};
```

Fig. 41. JAVA : The `Verify` class

```java
class MyData extends Verify{
  public void mymethod(){
    boolean B = false;
    eventually(B);
    B = true;
  }
}

class MyThread extends VThread{
  public void run(){
    boolean C = false;
    eventually(C);
    C = true;
  }
}

class Main extends Verify{
  public static void main(String[] args){
    boolean A = false;
    eventually(A);
    A = true;
    MyData mydata = new MyData();
    mydata.mymethod();
    MyThread mythread = new MyThread();
    mythread.start();
  }
}
```

Fig. 42. Java : Using the `Verify` class

14.4 Properties as Methods instead of as Statements

The solutions we have seen in the previous sections are based on expressing properties as JAVA statements that "get executed". The solution presented in this section is based on specifying properties by declaring methods having names that follow particular patterns. Hence, this solution keeps within the syntax of JAVA without extending the language and without requiring a particular parser.

```
class Verify{
  public static void assert(String s, boolean p){
    if (!p) System.out.println(s + " broken");
  }
}

class Counter extends Verify{
  public int x = 0;
  public void update(int dx){x = x + dx;};

  public void invariant(){assert("inv",x>=0);};

  public void pre_update(int dx)
    {assert("pre_update",x + dx >= 0);}

  public void post_update(int dx, int old_x){
    assert("post_update",x == old_x + dx);}
}

class Main{
  public static void main(String[] args){
    Counter counter = new Counter();
    int old_x;

    old_x = counter.x;
    counter.pre_update(-1);
    counter.update(-1);
    counter.post_update(-1,old_x);
    counter.invariant();
  }
}
```

Fig. 43. JAVA : Using methods to state properties

In the JAVA program shown in figure 43 we see a `Verify` class which basically just contains an `assert` method that will react appropriately if given a false argument. In the `Counter` class is given examples of how *property methods* can be defined, which not only will generate LTL formulae to be verified in SPIN, but which also can be called in the program to test the program in a traditional way. The `invariant` method defines an invariant we want to hold throughout the lifetime of the object. The methods `pre_update` and `post_update` define the pre and post conditions for the `update` method. There are many ways this can be expressed. In the presented solution a post-method takes as parameter all old variables referred to. There are at least two ways an invariant can be interpretated: either every use of the methods must be such that the invariant holds, or alternatively, if the methods are used correctly wrt. their pre-conditions, then the invariant can be guaranteed. One is a guide to the user, the other is a guarantee that can be locally proven correct.

When executing this program it will print:

```
pre_update broken
inv broken
```

Note that in a concurrent setting one will want to state rely/guarantee conditions, and this can be done in exactly the same way. However, in this case the pre and post conditions will contain temporal formulae stating: "*if it can be relied upon that the environment satisfies the temporal formula P then this method (object) can guarantee the behavior specified by temporal property Q*".

14.5 Comments and Language Extension

The approach we have seen elaborated has used the JAVA language itself for expressing temporal properties. Two alternatives is to either give specifications as comments or simply to extend JAVA. This is illustrated in figures 44 and 45. Of course in this case one can freely choose the syntax for formulae one wants, for example one closer to SPIN's LTL, even though that perhaps does not make specifications more readable. Explanatory names may be an advantage after all.

```
class Main{
  public static void main(String[] args){
    boolean A;
    boolean B;
/**
    specification{
      [property 1] <>A;
      [property 2] [](A -> <>B)
    }
*/
    A = true;
    B = true;
  }
}
```

Fig. 44. JAVA : Specifications as comments

```
class Main{
  public static void main(String[] args){
    boolean A;
    boolean B;

    specification{
      [property 1] <>A;
      [property 2] [](A -> <>B)
    };

    A = true;
    B = true;
  }
}
```

Fig. 45. JAVA : Extending JAVA

A The Producer/Consumer Example

```
/**********************/
/* System Definitions */
/**********************/

#define Index byte
#define ClassName byte
#define Producer  0
#define Consumer  1
#define CubbyHole 2
#define Main      3

#define true  1
#define false 0

#define undefined 0
#define MAX 5

#define null -1
#define this _pid

#define failure assert(0)
#define continue 0

typedef ObjRef{ClassName class; Index index};


/**************************/
/* Synchronization Locking */
/**************************/

#define get_LOCK(obj)
  CubbyHole_Obj[obj.index].LOCK

#define set_LOCK(obj,value)
  CubbyHole_Obj[obj.index].LOCK = value

#define lock(obj)
  atomic{
    get_LOCK(obj) == null ->
      set_LOCK(obj,this)}

#define unlock(obj)
  set_LOCK(obj,null)
```

```
/*******************/
/* Wait and Notify */
/*******************/

#define get_WAITING(obj)
  CubbyHole_Obj[obj.index].WAITING

#define set_WAITING(obj,value)
  CubbyHole_Obj[obj.index].WAITING = value

#define get_WAIT(obj)
  CubbyHole_Obj[obj.index].WAIT

#define wait(obj)
  atomic{
    unlock(obj);
    set_WAITING(obj,get_WAITING(obj) + 1);
    get_WAIT(obj)?continue;
    lock(obj)}

#define notify(obj)
  atomic{
    if
    :: get_WAITING(obj) > 0 ->
         get_WAIT(obj)!continue;
         set_WAITING(obj,get_WAITING(obj) - 1)
    :: else -> skip
    fi}

#define notifyAll(obj)
  atomic{
    do
    :: get_WAITING(obj) > 0 ->
         get_WAIT(obj)!continue;
         set_WAITING(obj,get_WAITING(obj) - 1)
    :: else -> break
    od}


/*******************/
/* class CubbyHole */
/*******************/

typedef CubbyHole_Class{
```

```
  int  LOCK;
  int  WAITING;
  chan WAIT = [0] of {bit};
  int  contents;
  bool available;
};
CubbyHole_Class CubbyHole_Obj[MAX];
Index CubbyHole_Next = 0;

#define CubbyHole_get_contents(obj)
  CubbyHole_Obj[obj.index].contents

#define CubbyHole_set_contents(obj,value)
  CubbyHole_Obj[obj.index].contents = value

#define CubbyHole_get_available(obj)
  CubbyHole_Obj[obj.index].available

#define CubbyHole_set_available(obj,value)
  CubbyHole_Obj[obj.index].available = value

#define CubbyHole_constr(obj)
  ObjRef obj;
  obj.class = CubbyHole;
  atomic{obj.index = CubbyHole_Next; CubbyHole_Next++};
  set_LOCK(obj,null);
  set_WAITING(obj,0);
  CubbyHole_set_available(obj,false)

#define CubbyHole_get_code(obj,return_value)
  do
  :: CubbyHole_get_available(obj) == false -> wait(obj)
  :: else -> break
  od;
  CubbyHole_set_available(obj,false);
  notify(obj);
  return_value = CubbyHole_get_contents(obj)

#define CubbyHole_get(obj,return_value)
  if
  :: get_LOCK(obj) == this ->
       CubbyHole_get_code(obj,return_value)
  :: else ->
       lock(obj);
       CubbyHole_get_code(obj,return_value);
```

```
      unlock(obj)
  fi

#define CubbyHole_put_code(obj,value)
  do
  :: CubbyHole_get_available(obj) == true -> wait(obj)
  :: else -> break
  od;
  CubbyHole_set_contents(obj,value);
  CubbyHole_set_available(obj,true);
  notify(obj)

#define CubbyHole_put(obj,value)
  if
  :: get_LOCK(obj) == this ->
      CubbyHole_put_code(obj,value)
  :: else ->
      lock(obj);
      CubbyHole_put_code(obj,value);
      unlock(obj)
  fi


/******************/
/* class Producer */
/******************/

typedef Producer_Class{
  ObjRef cubbyhole
};
Producer_Class Producer_Obj[MAX];
Index Producer_Next = 0;

#define Producer_get_cubbyhole(obj)
  Producer_Obj[obj.index].cubbyhole

#define Producer_set_cubbyhole(obj,value)
  Producer_Obj[obj.index].cubbyhole.class = value.class;
  Producer_Obj[obj.index].cubbyhole.index = value.index

#define Producer_constr(obj,c)
  ObjRef obj;
  obj.class = Producer;
  atomic{obj.index = Producer_Next; Producer_Next++};
  Producer_set_cubbyhole(obj,c)
```

```
proctype Producer_Thread(ObjRef obj){
  int i;
  i = 0;
  do
  :: !(i < 10) -> break
  :: i < 10 ->
       CubbyHole_put(Producer_get_cubbyhole(obj),i);
       i++
  od
};


/******************/
/* class Consumer */
/******************/

typedef Consumer_Class{
  ObjRef cubbyhole
};
Consumer_Class Consumer_Obj[MAX];
Index Consumer_Next = 0;

#define Consumer_get_cubbyhole(obj)
  Consumer_Obj[obj.index].cubbyhole

#define Consumer_set_cubbyhole(obj,value)
  Consumer_Obj[obj.index].cubbyhole.class = value.class;
  Consumer_Obj[obj.index].cubbyhole.index = value.index

#define Consumer_constr(obj,c)
  ObjRef obj;
  obj.class = Consumer;
  atomic{obj.index = Consumer_Next; Consumer_Next++};
  Consumer_set_cubbyhole(obj,c)

proctype Consumer_Thread(ObjRef obj){
  int value;
  int i;
  int received[10];
  value = 0;
  i = 0;
  do
  :: !(i < 10) -> break
  :: i < 10 ->
```

```
      CubbyHole_get(Consumer_get_cubbyhole(obj),value);
      received[i] = value;
      i++
  od;
  i = 0;
  do
  :: !(i < 10) -> break
  :: i < 10 ->
      assert(received[i] == i);
      i++
  od;
};


/**************/
/* class Main */
/**************/

init{
  CubbyHole_constr(c);
  Producer_constr(prod,c);
  Consumer_constr(cons,c);
  run Producer_Thread(prod);
  run Consumer_Thread(cons);
}
```

References

1. Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.
2. Mary Campione and Kathy Walrath. *The Java Tutorial, Object-Oriented Programming for the Internet*. Addison Wesley, 1996.
3. Klaus Havelund, Mike Lowry, and John Penix. Formal Analysis of a Space Craft Controller using SPIN. Technical report, NASA Ames Research center, Moffett Field, California, USA, August 1997.
4. Gerard Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.
5. Doug Lea. *Concurrent Programming in Java*. Addison Wesley, 1997.
6. B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan Execution for Autonomous Spacecrafts. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, 1997.