5 Synchronization

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 = у + d;}; |
| public synchronized void | add(int dx, int dy){ |
| int old\_x = x; |  |
| int old\_y = y; |  |
| add2x(dx); |  |
| add2y(dy); |  |
| Yerify.assert("updated1 | , x == old\_x + dx к |
|  | у == old\_y + dy); |
| } |  |
| } |  |

Fig. 11. **Java** : methods add2y and add become synchronized

1. Translation
2. 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\_L0CK and set\_L0CK 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\_0bj[obj.index].LOCK

#define set\_L0CK(obj.value)

XY\_0bj[obj.index].LOCK = value

#define lock(obj) atomic{

get\_LOCK(obj) == null -> set\_LOCK(obj.this)}

#define unlock(obj)

**set\_L0CK(obj,null)**

1. 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 у

}:

XY\_Class XY\_0bj[MAX];

Index XY\_Next = 0;

#define XY\_get\_y(obj)

XY\_0bj[obj.index].у

#define XY\_set\_y(obj.value)

XY\_0bj[obj.index].у = value

#define XY\_constr(obj)

ObjRef obj; obj.class = XY;

atomicjobj.index = XY\_Next; XY\_Next++}; set\_L0CK(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\_L0CK(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 к XY\_get\_y(obj) == old\_y + dy)

#define XY\_add(obj,dx,dy) if

:: get\_L0CK(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 Notify All

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

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 waitQ and notifyQ 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 waitQ and notifyO 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 = с;}

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 ();

}

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

Fig. 17. Java : An example program - part II

1. Translation

In this section we shall explain those parts of the translation that are related to the wait () and notif у () 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 model checker. 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 ‘==’):

Verify.assert(”received ok”,received[i] == 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;

h

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 CubbyHolejQbj 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 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) atomicj

unlock(obj);

set.WAITING(obj,get.WAITING(obj) + 1);

get.WAIT(obj)?continue;

lock(obj)}

«define notify(obj) atomicj if

:: get.WAITING(obj) > 0 -> get.WAIT(obj)!continue; set.WAITING(obj,get.WAITING(obj) - 1) :: else -> skip fi)

«define notifyAll(obj) atomicj 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.

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 .printlnC'assertion " + name + " broken");

}:

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

public static void write(String s, boolean b){

System.out.println(s + b);

}:

h

Fig. 21. **Java** : Extending the “logic”

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 “о (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.

1. The Java Program Now to the Java program itself, see figure 22. It shows two how two threads, al and a2 of types Agent 1 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.

Agent 1 basically assigns true to Status.a, and then suspends itself. Agent2 assigns true to Status.b and then resumes Agent 1. From the main method it can be seen how the Agent2 thread is initialized with the Agent 1 thread (al) as parameter. When Agent 1 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.с will become true” (meaning that Agent 1 terminates), and “whenever Status.с is true, so is Status.a and Status.b” (meaning for example that Agent2 has executed - to resume Agent 1).

class Status{

public static boolean a = false;

public static boolean b = false;

public static boolean с = false;

}:

class Agent1 extends Thread{ public void run(){

Status.a = true;

Verify.write("a == ".Status.a); suspend();

Status.с = true;

Verify.write("с == ".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 al = new Agent1();

Agent2 a2 = new Agent2(al); al.start(); a2.start();

Verify .assert **("[] (с => a** к **b)",**

!Status.с II (Status.a к Status.b));};

}

Fig. 22. **Java** : An example program

1. Translation
2. 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 Procld = 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 Agentl and Agent2, and in particular the definition of the typedefs Agentl.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.

#define set\_PID(obj.value) if

:: obj.class == Agentl -> Agentl\_Obj[obj.index].PID = value :: obj.class == Agent2 -> Agent2\_0bj[obj.index] .PID = value fi

#define running ! suspended [this]

#define suspend(obj) if

:: obj.class == Agentl ->

suspended[Agentl\_Obj[obj.index] .PID] = true :: obj.class == Agent2 ->

suspended[Agent2\_0bj[obj.index].PID] = true :: failure fi

#define resume(obj) if

:: obj.class == Agentl ->

suspended[Agentl\_Obj[obj.index].PID] = false :: obj.class == Agent2 ->

suspended[Agent2\_0bj[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

1. 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.

1. Translation of Class Agentl The translation of class Agentl, 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 с = false;

}:

Status\_Static Status\_Static\_Area;

Fig. 25. **Spin** : Translation of class Status

object of type Obj Ref 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 Agentl\_Class{ Procld PID }; Agentl\_Class Agentl\_Obj[MAX] ; Index Agentl\_Next = 0;

#define Agentl\_constr(obj)

ObjRef obj;

obj.class = Agentl;

atomicjobj.index = Agentl\_Next; Agentl\_Next++}

proctype Agentl\_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.с = true; write(Status\_Static\_Area.c);

h

Fig. 26. **Spin** : Translation of class Agentl

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.

1. **Example Java Program to be Translated**
2. 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.

class Obj{

public static Data data;

public static Job job;

public static Stop terminator;

public static boolean finalized = false;

}:

class Data{

public synchronized void workO { while (true) {/\* do some job\*/};

}:

public synchronized void finalize(){

Obj.finalized = true;

}:

}

class Job extends Thread} public void run(){

Obj .data.workO ;

}; }:

class Stop extends Thread} public void run()}

Obj. job.stopO ;

Verify.eventually(Obj.finalized);

}; }:

class Main}

public static void main(String[] args)}

Obj.data = new DataO ;

Obj.job = new Job();

Obj .terminator = new StopO ;

Verify.eventually(Obj.finalized);

Obj.job.start();

Obj.terminator.start();

try}0bj . job. joinQ ;}catch(InterruptedException e)}}; Verify .assert ("job has stopped",! Obj . job. isAliveO); Obj.data.f inalize();

}:

h

1. **Translation**
2. 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 Procld = 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\_L0CK and set\_L0CK just reads and writes to the LOCK field as we have seen before.

The macros add\_LOCKING, remove\_LOCKING and clear\_L0CKING 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 JLOCKING 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\_L0CK(obj.value)

Data\_0bj[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\_0bj[obj.index].PID = value

:: obj.class == Stop -> Stop\_0bj[obj.index].PID = value

:: failure fi

#define get\_PID(obj)

(obj.class == Job -> Job\_0bj[obj.index].PID :

(obj.class == Stop -> Stop\_0bj[obj.index].PID : undefined))

#define set\_STOP(obj,value) if

:: obj.class == Job -> Job\_0bj[obj.index].STOP = value

:: obj.class == Stop -> Stop\_0bj[obj.index].STOP = value

:: failure fi

#define get\_ST0P(obj)

(obj.class == Job -> Job\_0bj[obj.index].STOP :

(obj.class == Stop -> Stop\_0bj[obj.index].STOP : undefined))

#define isAlive(obj)

(get\_PID(obj) != null к 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)}

1. 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 С is the class name. Observe that the constructor is called Datajiew 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 Datamew(data).

typedef Data\_Class{

Procld LOCK = null

}:

Data\_Class Data\_0bj [MAX] ;

Index Data\_Next = 0;

#define Data\_new(obj) obj.class = Data;

atomicjobj.index = Data\_Next; Data\_Next++}

#define Data\_work\_code(obj) do

:: skip od

#define Data\_work(obj) if

:: get\_L0CK(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\_L0CK(obj) == this ->

Data\_finalize\_code(obj)

:: else ->

lock(obj);

Data\_finalize\_code(obj); unlock(obj)

fi

9