On alien call & connecting virtual machines

concurrent and distributed programming have one model

Andrzej Salwicki

Chair of Informatics UKSW, formerly at Institute of Informatics UW

October 1, 2010

Our plan

Introduction

We present an original protocol of cooperation among active objects. It differs from the other approaches: monitors, rendez-vous, message passing. However, all known mechanisms of synchronisation and/or communication among processes can be easily defined by the proposed protocol.
The second part of our message says: concurrent and distributed programming can be done in a uniform way. Remark, that in most cases the tools used for concurrent programming differ from the tools of distributed programming. Our approach makes the process of programming easier.
The ideas we present are of general and universal character. They may be adapted in various environments. The methodology of programming active objects was validated by an implementation in Loglan“82 programming language. Consequently the examples will be given in this language.

Terminology

Before going into details we need to fix the notions.

thread – a seqence of instructions to be executed, it is a property of

active object – any object equipped with a nonempty sequence of instructions, in Java they are contained in the method *run*.

Assumptions I

We assume that programs and systems of programs are written in one object-oriented programming language $\mathcal{L}$. Next, we assume that the language $\mathcal{L}$ admits one predefined class **process** (The name is of no importance, one may prefer the name ActiveObject). The objects of this class and of classes derived from it will be called *active objects*. Each active object o has the following properties:

- object o has a thread i.e. a list of instructions to be executed,
- object o is either in *passive* state or in *active* state,
- the object may enter an active state (In the state the instructions of the thread are executed concurrently with the instructions of other threads. The main program is also a thread.)
- the active object may enter a passive state (e.g. when the command suspend() is executed),
- each method of the object is either *enabled* or *disabled*. Initially all the methods of any active object are disabled.

Assumptions II

- instruction enable ⟨list of methods⟩ causes that all the methods from the list become enabled. (One may believe the enabled methods are public.)
- instruction disable ⟨list of methods⟩ causes that all the methods from thelist become disabled. (Analogously, one may believe the disabled methods are private). The status of a method may change from disabled to enabled to disabled .etc. ...
- instruction accept is a point of rendez-vous with an instruction calling a method of this process...

Assumptions III

- a calling instructuction - has the form
  call o.method(...)
  and is executed in cooperation with a corresponding instruction accept in the thread of process o (synchronous alien call) or
  interrupts the execution of the called process o (asynchronous alien call). The asynchronous alien call occurs when the method m is enabled in process o.
  Alien call differs from calling a method in a monitor object. In Java a call of the method which belongs to an object (monitor) o is executed by the thread of caller. Here we postulate that it is a callee which is to execute the method. It implies that the two processes: the caller and the callee have to cooperate.

Scenario of active object I

Creation of an active object:
Elaboration of the object expression `new MyProcess(parameters)` returns an object *o* of `type Myprocess`. Hence, the execution of the assignment instruction

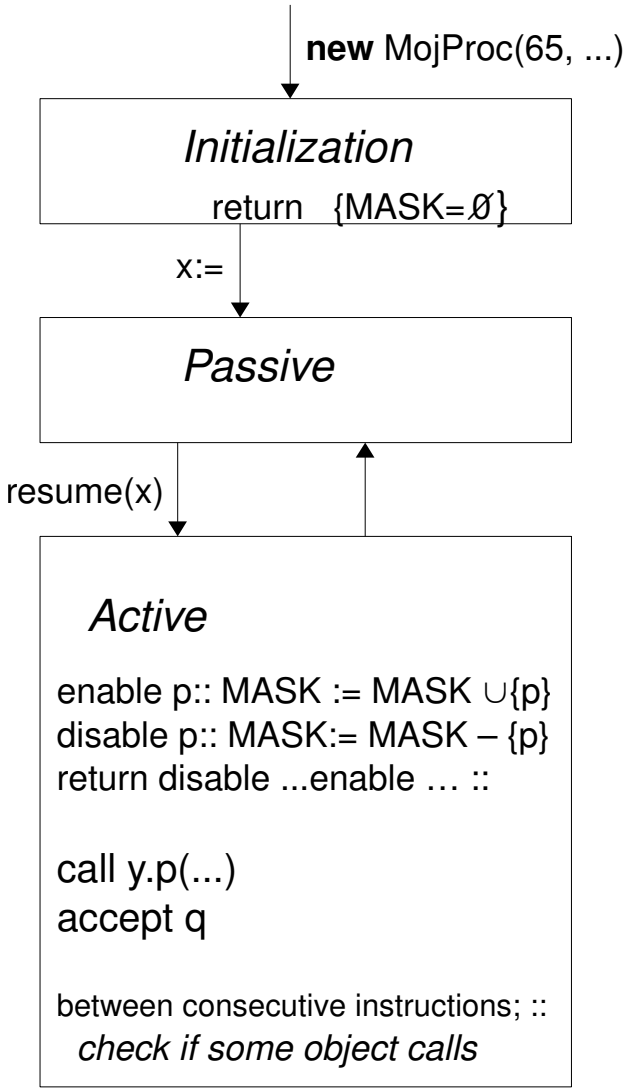
```
z := new MyProcess(parameters)
```

leads to a new configuration where, the set of existing objects is augmented by the object *o*, object *o* is the value of the variable z. One may say also, the object *o* is pointed out (is referenced to) by the variable z. Remark, the object may be allocated on one computer and the variable z may be on another computer. An active object *o* after it has been created, remains in the state Passive. Another active object, owner of variable z, such that the value of z is the object o may activate the . Object *o* of name *z* becomes Active when another object executes the command `resume(z)`. An active object may execute command `suspend()` and enter the state passive.

Scenario of active object II

Termination: an active object may reach the end of its thread, for example it may reach `end Myprocess`. In this case the active object is killed and deallocated. For the object cannot be activated again and its resources being private will never be accessible from outside.
Awaiting – an active object may enter the state Awaiting if it awaits for a partner object to jointly execute a procedure instruction. It happens if either the current object begins execution of alien call of a procedure or if the object begins execution of the instruction accept (see below).

Scenario of active object III

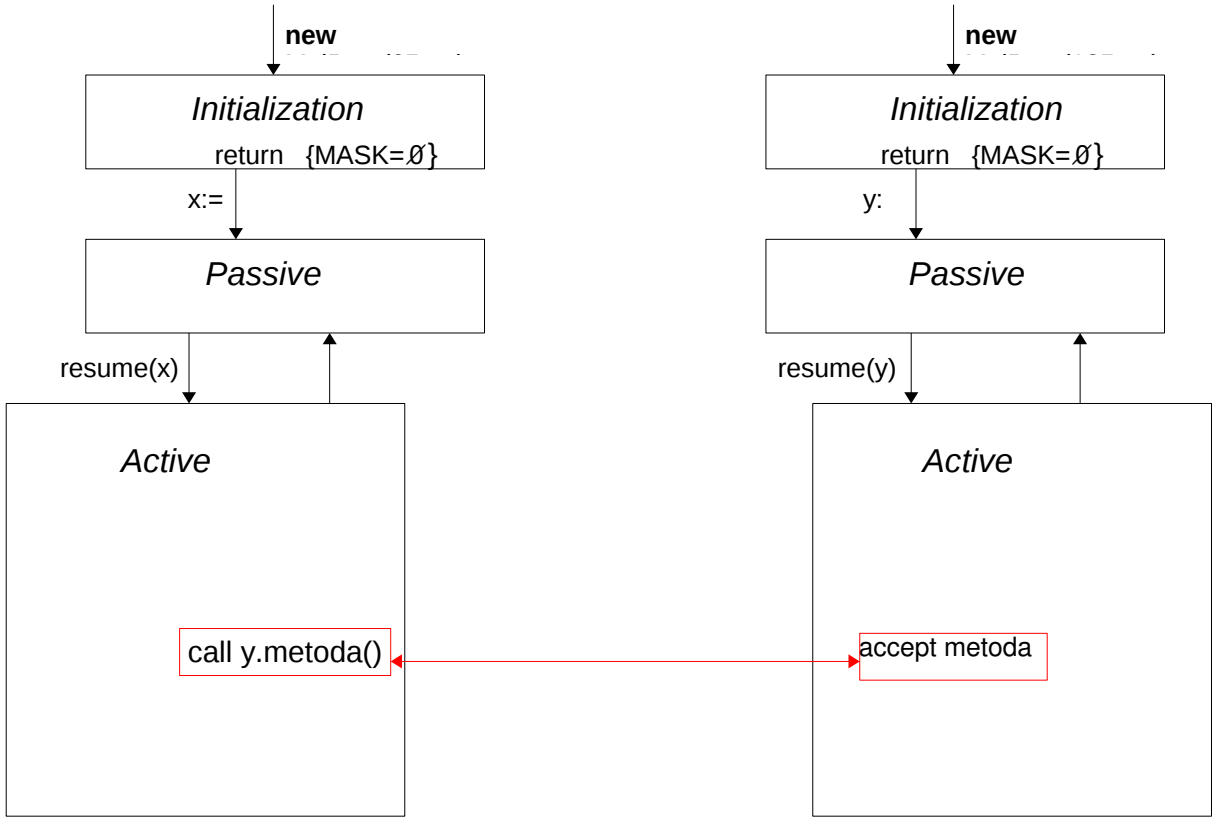


Rysunek: The scenario of active object

Alien call protocol I

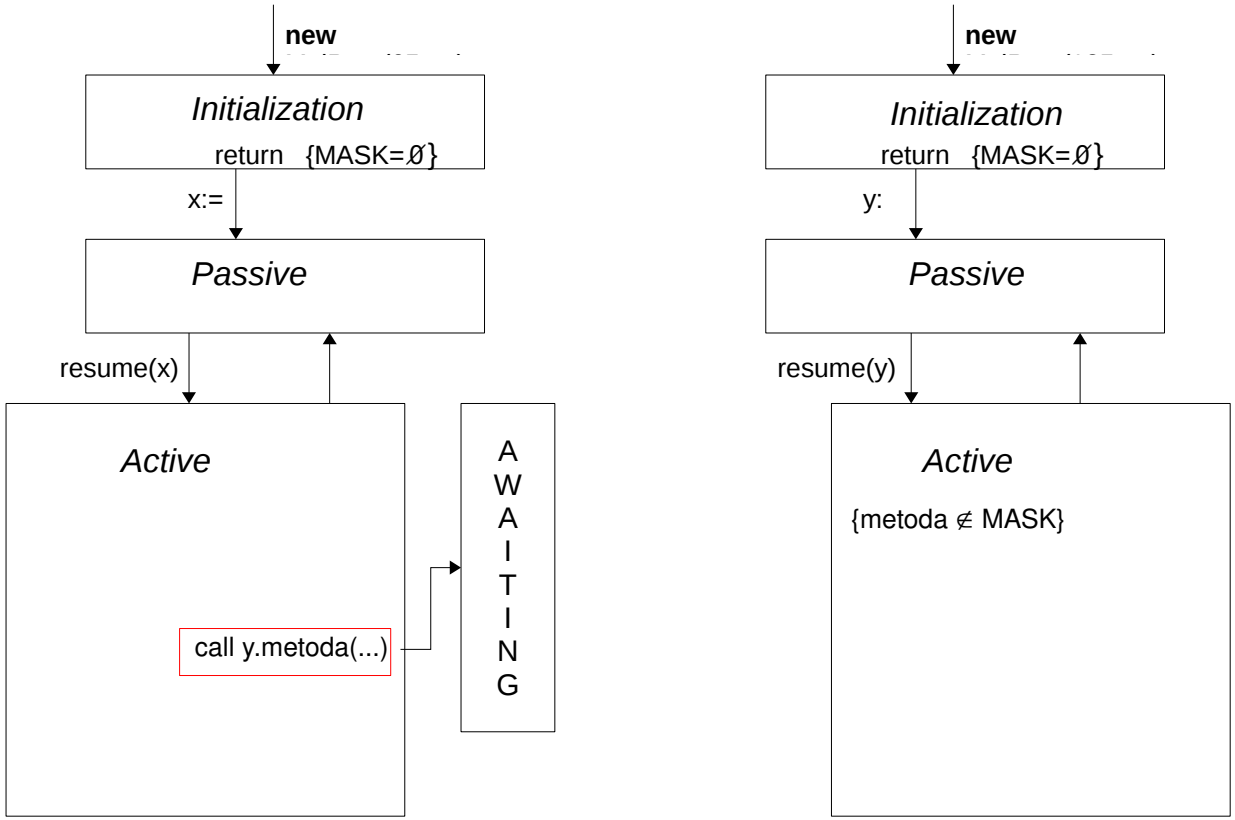
We shall illustrate the protocol by a series of pictures.

Alien call protocol II



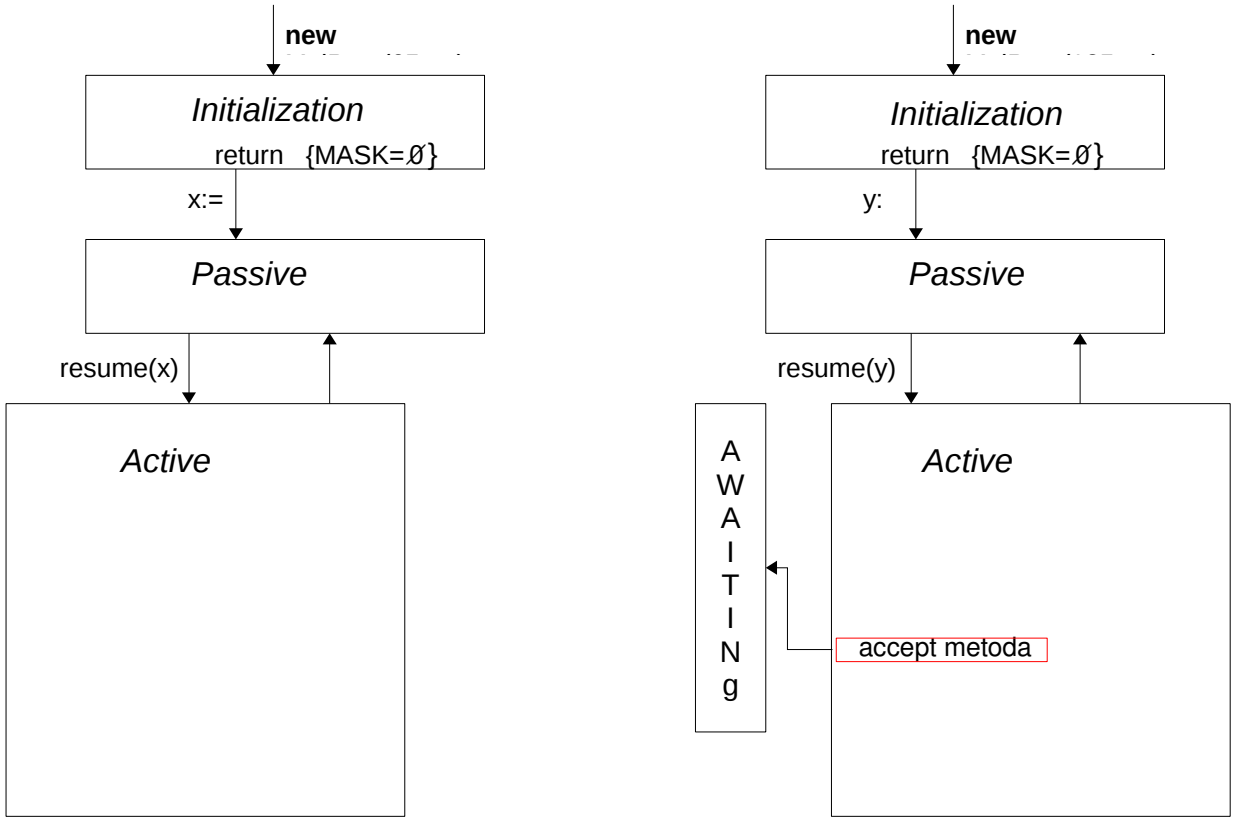
Rysunek: Protocol of alien call part1

Alien call protocol III



Rysunek: Protocol of alien call part 2

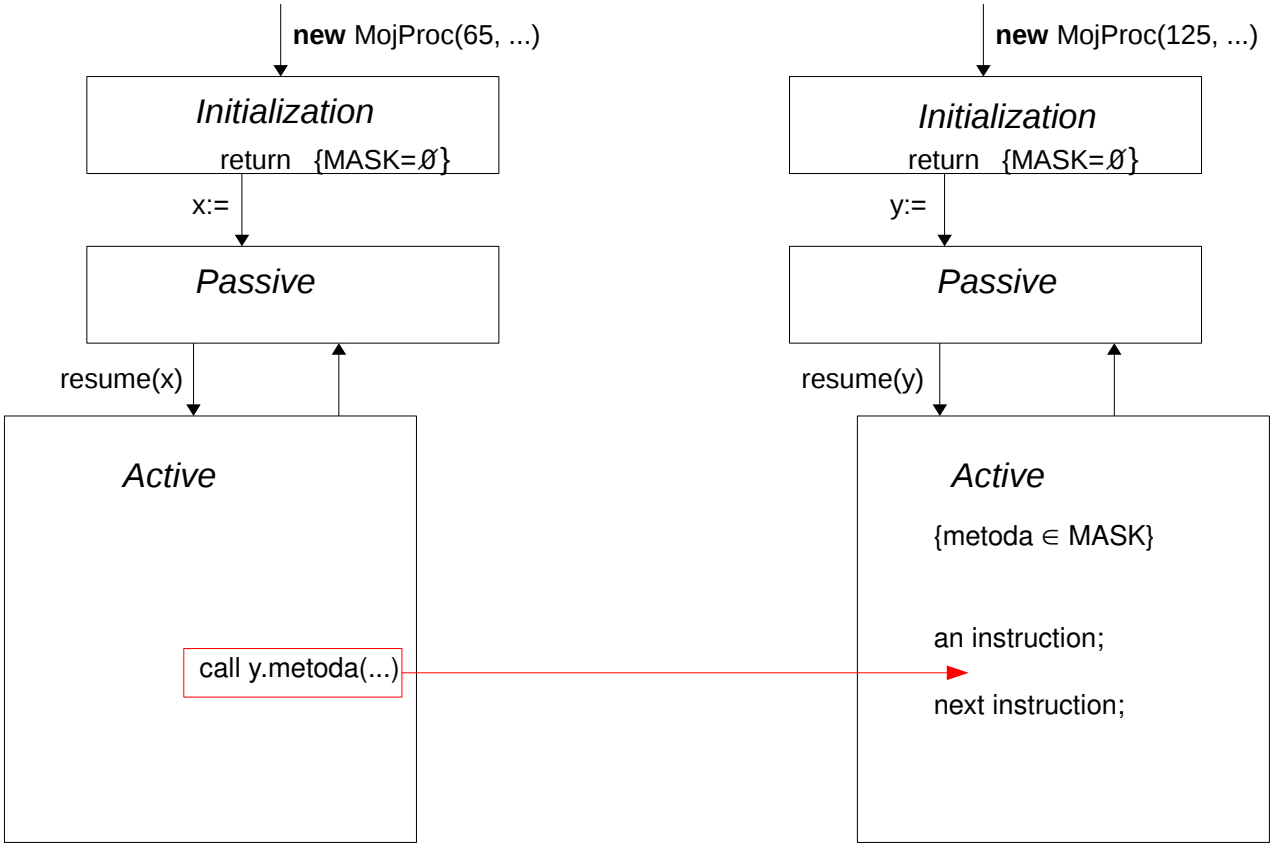
Alien call protocol IV



During execution of instruction accept: MASK (of thread y) := MASK∪ {metoda}.
While no active object execute instruction „call y.metoda;", active object y is awaiting.

Rysunek: Protocol of alien call part 3

Alien call protocol V



Asynchronous case: when active object x calls metoda in y and metoda ∈ MASK, the thread of active object y is interrupted, object y executes method metoda and returns to its own thread.

Rysunek: Protocol of alien call part 4

Properties I

In this section we attempt to describe the properties of active objects from the point of view of a user.

P1 An active object is created and memorized when an assignment instruction `z:=new Myprocess()` is executed. Note, creation of an active object without assignment has no sense, for the newly created object will become a garbage immediately.

P2 The newly created object will be allocated on a computer indicated by the value of the first parameter. The value 0 tells that new active object will be allocated and run on the same computer (concurrency).

Properties II

P3 **Mutual exclusion**. If several active objects simultaneously execute alien procedure calls of one active object o (a callee), then only one at the time may execute it.
The following algorithmic formula expresses the mutual exclusion of n parallel alien calls. The formula abstracts from the possible other threads.
$\Box \parallel_{i=1}^{n} [o.m_{i_j}; R_i]\alpha \Leftrightarrow$
$\bigvee_{k=1}^{n} \Box \overline{o.m_{k_j}}; [\parallel_{i=1, i\neq k}^{n} [o.m_{i_j}; R_i] \parallel R_k]\alpha$ The expression $\overline{o.m_{k_j}}$ denotes the body of the method m in object o, modified by the actual parameters.

P4 **Dynamic public/private methods**. Each method of an active object may be public in one moment and private in another one. A method m is public when its name is in the mask of the object, when its name does not belong to the mask, the method is private.

Properties III

P5 When one active object calls a method *m* of another active object and the method *m* is in the mask then we have an effect of *interruption*. The callee interrupts its own work and executes a service for the caller object.

P6 When one active object calls a method *m* of another active object and the method *m* is not in the mask and the callee executes the instruction accept with the name *m* on the list, then we have an effect of *meeting*. The caller object and the callee object meet and execute the called method jointly.

P7 Each active object may be once a client calling a procedure in a remote active object and in another moment it can be a server, ready to serve one of its procedures to other active objects.

P8 **Distributed concurrency is true fair concurrency**.
The programs like [p:=false || while p do x := x+1] always terminate.

Example 1

We begin with an example showing a couple of threads printing in the screen.

```
program first;
    unit writer: process(node: integer, c: char);
        var i: integer;
    begin
        return;

        for i := 1 to 32 do
            write(c);
        od
    end writer;
var w1, w2, w3: writer;
begin
    w1 := new writer(0, 'a'); w2 := new writer(0, 'b');
    w3 := new writer(0, 'c');
    resume(w1); resume(w2); resume(w3);
end
```

Remarks I

Perhaps you guessed the image on the screen shows a mixture of letters 'a', 'b' amd 'c'. This is so because the threads w1, w2 and w3 compete for the screen – their common resource. From this example we learn that:

1. processes, aka active objects, are created dynamically, by means of **new** expression,
2. instructions of a process (after **begin**) divide to constructor and thread. Instructions of constructor terminate with **return** instruction, instructions appearing after return form a thread.
3. the first parameter of new, equal 0 indicate that active object will be allocated on the same virtual machine.

Example 2 - binary semaphore I

Our next example shows how to define a semaphore, the tool for synchronization of processes. A semaphore *s* is an active object with three methods: pass, free and fin. The methods have empty bodies. The thread of the semaphore *s* repeats instructions `accept pass,fin; accept free, fin;` until one of clients execute the command `call s.fin`. If all clients follow the same scheme `call s.pass; critical section; call s.free`, then no interleaving of commands of critical sections is posible.
allowframebreaks

```
program Second;
    unit binarySemaphore: process(node:integer);
    unit pass: procedure;
    end pass;
    unit free: procedure;
    end free;
    unit fin: procedure;
    begin
        bol := false;
```

Example 2 - binary semaphore II

```
    end fin;
    var bol: boolean;
begin
    bol := true;

    return;
    enable fin;
    while bol do
        accept pass;
        accept free
    od;
end binarySemaphore;
unit writer: process (node:integer, nr:integer, c: char, sem: aSemaphore);
    var i: integer;
    unit fin: procedure;
    begin
    end fin;
begin
    return;
```

Example 2 - binary semaphore III

```
    call sem.pass;
    for i := 1 to 12 do
        write(a(i));
    od;
    writeln;
    call sem.free;
    accept fin;
end writer;
var s: aSemaphore, w1, w2, w3: writer, i: integer;
begin
    s := new aSemaphore(0);
    resume(s);
    w1:= new writer(0,1,'a',s);
    w2:= new writer(0,2,'b',s);
    w3:= new writer(0,2,'c',s);
    writeln("press Enter");
    readln;
    resume(w1);
```

Example 2 - binary semaphore IV

```
    resume(w2);
    resume(w3);
    call w1.fin; call w2.fin; call w3.fin;
    call s.fin;
end Second
```

Spooler example I

The following example is more interesting. We shall analyse it more closely.

```
unit queue:class(type element; size:integer);
    (* The auxiliary class implementing
    queues with a limited capacity.
    The class is parameterized by the element
    type and the maximum queue size *)

    unit insert:procedure(e:element); ...
        (* insert element into the queue *)
    unit delete:function:element; ...
        (* remove the first element *)
    unit isempty:function:boolean; ...
        (* check if the queue is empty *)
    unit isfull:function:boolean; ...
        (* check if the queue is full *)

end queue;
...
```

Spooler example II

```
unit spooler:process;
    var Q:queue, (* queue of files *) f:filename;
    unit print:procedure(f:filename);
    begin
        call Q.insert(f);
        if Q.isfull
        then
            return disable print
        fi;
    end print;
begin
    Q := new queue(filename, 50);
    return;
    do
        disable print;
        if Q.isempty
        then accept print
            fi;
            f := Q.delete;
```

Spooler example III

```
        enable print;
        (* send the file f to the printer *)
        ...
    od
end spooler;
```

Two questions arise:

1. Suppose several active objects simultaneously require printing by executing

   call s.print(f) || call s.print(f')

   commands in two objects p and q. Can we assure that no request will be lost or improperly queued?
2. Suppose that the spooler takes a file from the queue to be sent to a printer and simultaneously one or more processes require printing of their files. Can we assure that files will be printed without interleaving their contents and in proper order?

Spooler example IV

These questions find the following answers.

Lemma

No request will be lost and the requests will be handled as first-in first-out policy requires.

The positive answer to the first question is founded on the alien call protocol. For it will be impossible that two activation records of procedure print coexist. As concerns the second question: it is sure that the operation Q.delete will not interfere with any operation Q.insert. It is so because we disabled the operation print before attempting to execute operstions delete and isempty.

Specify the alien call protocol

The alien call protocol was invented by Bolek Ciesielski in 1988. It was never published in a scientific journal.

How it works

Diagrams

Properties

Progress

Do it in Java

Is it possible to implement the alien call protocol in Java?
At the beginning we thought *it is easy*. Now we are not so sure.

Progress

Specification of ActiveObject class I

Problem: write a class ActiveObject in Java which implements the methods enable, disable, accept as described earlier.
Specification - Java style follows

```
interface ActiveObjects extends Runnable  {
    /* initially, a new ActiveObject is passive, the set of Enabled
        methods is empty. */
    /* methods changing status */
    void resume(SpecifiAO o)
    void stop()
    /* the instruction stop hangs the execution of the thread,
        the instruction resume(o) resumes execution of the thread o. */
    /* methods changing Mask */
    void enable(String m)
    void disable(String m)
    /* methods of collaboration */
```

Specification of ActiveObject class II

void accept(String m)
void alienCall()
/* Let ActiveObjects be a class implementing this specification.
Let P be a class extending the class ActiveObjects.
Let o be an active object of the class P.
Consider the instructions of the thread o.

A) the effect of enable: the methods m of the thread become enabled.
Enabled := Enabled $\cup$ {m}
B) the effect of disable
Enabled := Enabled $\setminus$ {m}
C) the effect of accept
The instruction accept will be executed in cooperation of an instructio
alienCall, see the point E.
It means that another active object of
a class derived from ActiveObject must

Specification of ActiveObject class III

begin to execute
alienCall(o, m, params)

D) the effect of an alien call
D1) the object o must be in state Active,
the method m must be Enabled.

E) When a rendez-vous of two Active Objects
is reached then
e1) the parameters of alien call of the
method m are transferred to the object o,
e2) the called object o executes the method m with with the actual parametersobtained from the caller object.

F) Asynchronous alien call
When one caller process encounter an alien call instruction and the callee process is active and the method is enabled then the callee interrupts the execution of its

Specification of ActiveObject class IV

```
    thread and executes alien call instruction
    - see E)
*/
} // end of interface
```

Progress

Remarks

Let us describe which solutions should be rejected.
Any solution that requires a modification of JVM - Java Virtual Machine. For it means a modification of the language. More important it will be rejected by the users. Any solution which requires that program is to be edited after compilation. We can imagine two attempts to implement the protocol:

1. The first attempt would consist in replacing semicolons by calls of an instruction checking whether an alien call occurred.
2. Another concept would consist in editing the file .class after compilation. One may insert the calls of a method which checks whether an alien call occurred.

Someone suggested to use a listener object. Is it a solution?

Prize

An author of a solution or of a proof that no solution exist will obtain a prize. Today, October 1, 2010, the prize is 100 Euro.

Literature