Comparing Java, C# and Ada Monitors queuing policies : a case study and its Ada refinement

Claude Kaiser, Jean-François Pradat-Peyre, Sami Évangelista, Pierre Rousseau

CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris
{kaiser, peyre, evangeli, rousseau}[at]cnam[dot]fr
http://quasar.cnam.fr

Abstract. Learning concurrency paradigms is necessary but it is not sufficient since the choice of run-time semantics may introduce subtle programming errors. It is the aim of this paper to exemplify the importance of process queuing and awaking policies resulting from possible choices of the monitor concept implementation.

The first part of the paper compares the behaviour of concurrent processes sharing a unique waiting queue for condition synchronization when implemented in Java or in Ada. A particular solution of the dining philosophers paradigm will be used to show how the difference in the monitor semantics may lead or not to deadlock. This comparison provides insight for deriving a correct Java implementation. The second part of the paper shows how the implementation can be refined when using Ada entry families and requeue with requeue once restriction. The result is elegant, safe and fair, and deterministic. This paper ends with quantitative comparisons of concurrency complexity and of concurrency effectiveness.

We conclude that Java and C# multithreading need defensive concurrent programming while Ada allows more latitude for developing correct concurrent programs.

1. Introduction

Concurrent programming is still challenging and difficult. "*Since concurrency techniques have become indispensable for programmers who create highly available services and reactive applications, temporal dimensions of correctness introduced by concurrency, i.e., safety and liveness, are central concerns in any concurrent design and its implementation*" [Lea 98]. And without expert guidance and concurrent design-pattern description, they're expected to occasionally fail. Thus providing significant examples and paradigms for teaching good and correct style is of prime importance.

Learning concurrency paradigms is necessary but it is not sufficient. The choice of the run-time semantics must be known since it may introduce subtle design and programming errors. It is the aim of this paper to exemplify the importance of process queuing and awaking policies (whether processes are named threads or tasks) resulting from possible choices of the monitor concept implementation.

The languages Java, C# and Ada implement the monitor concept [Hoare 1974]. Several possible monitor concurrency semantics have been used in the past and a classification is presented in

[Buhr1995]. Every implementation provides mutual exclusion during the execution of a distinguished sequence (synchronized method in Java, lock in C#, protected object subprograms in Ada) using a lock for every object. The semantics differ in the chosen policies for blocking, signalling and awaking processes.

The Java policy uses explicit self-blocking and signalling instructions. It provides "wait()","notify()" and "notifyAll()" clauses with a unique waiting queue per encapsulated object (termed "synchronized"). A self-blocking thread joins the waiting queue and releases the object mutual exclusion lock. A notifying thread wakes up one or all waiting threads (which join the ready threads queue), but it does not release the lock immediately. It keeps it until it reaches the end of the synchronized "method" (or "block") ; this is the "signal and continue" monitor discipline.

Hence the awaken threads must still wait and contend for the lock when it becomes available. However, as the lock is released, and not directly passed to an awaken thread (the lock availability is globally visible), another thread contending for the monitor may take precedence over awaken threads. More precisely, as the awaken threads share the ready queue with other threads, one of the latter may take precedence over the formers when contending for the processor; if this elected thread calls also a synchronized method (or enters a synchronized block) of the object, it will acquire the lock before the awaken threads and then access the object before them. This may contravene the problem specification and may require the use of defensive programming.

Java 1.5 allows using multiple named condition objects. This provides more programming flexibility, however the signalling policy remains the same.

The language C# has thread synchronization classes which expressiveness is close to Java 1.5 using for example Wait(), Pulse(), Monitor.Enter(), Monitor.Quit() and which queuing and signalling semantics is similar. Thus we shall refer to Java examples only.

Ada provides protected object types which has no low level clauses for blocking and awakening tasks. Condition synchronization relies on programmed guards (a boolean expression termed "barrier"). Access is provided by calling entries, functions and procedures, but only one of these can be executed at a time in mutual exclusion. The entries have barrier conditions, which must be true before the corresponding entry body can be executed. If the barrier condition is false, then the call is queued and the mutual exclusion is released. At the end of the execution of an entry or a procedure body of the protected object, all barriers which have queued tasks are re-evaluated and one waiting call which barrier condition is now true is executed. The mutual exclusion is released only when there is no more waiting task with a true barrier condition. Thus existing waiting calls with true barrier condition take precedence over new calls. This is the "eggshell model" for monitors. Evaluation of barriers, execution of protected operation sequences, and manipulation of entry queues are all done while the lock is held.

The "requeue" statement enables a request to be processed in two or more steps, each associated with an entry call. The effect is to return the current caller back to an entry queue. The caller is neither aware of the number of steps nor of the requeuing of its call. This sequence of steps corresponds to a sequential automaton. According to the eggshell model, any entry call of such a sequence which guard has become true has precedence over a new call contending for the protected object.

If we summarize the differences in the awaking policies: Java and C# make no choice and leave all ready processes able to compete for the lock, while Ada gives precedence to the processes waiting in entry queues.

The first part of this paper compares the behaviour of concurrent processes sharing a unique waiting queue for condition synchronization when implemented in Java or in Ada. A particular solution of the dining philosophers paradigm will be used to show how the difference in the monitor semantics, i.e. in the awaking policies, may lead or not to deadlock. This comparison provides insight for deriving a correct Java implementation.

The second part of the paper shows how the implementation can be refined when using Ada entry families and requeue with requeue once restriction. The result is elegant, safe and fair, and deterministic.

This paper ends with quantitative comparisons of concurrency complexity and of concurrency effectiveness. We conclude that Java and C# multithreading need defensive concurrent programming while Ada allows more freedom for developing correct concurrent programs.

2. A new solution for the dining philosophers paradigm

The dining philosophers, originally posed by Dijkstra [Dijkstra 7I], is a well-known paradigm for concurrent resource allocation. Five philosophers spend their life alternately thinking and eating. To dine, each philosopher sits around a circular table at a fixed place. In front of each philosopher is a plate of food, and between each pair of philosophers is a chopstick. In order to eat, a philosopher needs two chopsticks, and they agree that each will use only the chopsticks immediately to the left and to the right of his place. The problem is to write a program simulating the philosopher's behaviours and to devise a protocol that avoids two unfortunate conclusions. In the first one, all philosophers are hungry but none is able to acquire both chopsticks since each holds one chopstick and refuses to give it up. This is deadlock, a safety concern. In the second one, a hungry philosopher will always lack one of the two chopsticks which are alternately used by its neighbours. This is starvation, a liveness consideration.

This paradigm has two well-known approaches for obtaining a solution. In the first one, the chopsticks are allocated one by one, and a reliable solution is obtained by adding one of the usual constraints for deadlock prevention: the chopsticks are allocated in fixed (e.g., increasing) order; a chopstick allocation is denied as soon as the requested allocation would lead to an unsafe state (seated dinner, with only 4 chairs). Ada implementation of this approach can be found in [Burns 1995, Barkaoui 1997]. In the second one, the chopsticks are allocated globally only, which is a safe solution; when a fair solution is necessary, it is obtained by adding reservation constraints, care being taken that these constraints do not reintroduce deadlock. Ada implementation are given in [Brosgol 1996, Kaiser 1997]

Let us consider now another approach, which does not seem to have been much experimented except in [Kaiser 1997]. The chopsticks are allocated as many as available and the allocation is completed as soon as the missing chopsticks are released. Let us observe the behaviour of this solution when implemented in Java and in Ada and from these experiments, let us determine the conditions of its correctness.

3. Comparing Java and Ada monitor semantics with this new solution

This section presents several implementations which, influenced by the Java style, use all a unique waiting queue, implicitly and compulsory in Java, explicitly and restrictively in Ada:

- A Java class with synchronized methods which are dependant of the Java monitor semantics,
- A transcription of this Java class into an Ada protected object associated with embedding procedures used to skirt the Ada eggshell semantics, in order to force it to behave as a Java monitor,
- An Ada regular implementation of a protected object using fully the Ada eggshell semantics.

3.1. Java implementation

The Java implementation style is influenced by the choice of a monitor semantics with a unique implicit waiting queue. It leads to the following Chop class with get_LR and release methods.

```
public final class Chop {

    private int N ;
    private boolean available [ ] ;

    Chop (int N) {
        this.N = N ;
        this.available = new boolean[N] ;
        for (int i =0 ; i < N ; i++) {
            available[i] = true ; // non allocated stick
        }
    }

    public synchronized void get_LR (int me) {
        int score = 0 ; // pseudo program counter used for transcripting the Java code in Ada
        while ( !available [me]) {
            try { wait() ; } catch (InterruptedException e) {}
        }
        available [me] = false ; score = 1; // left stick allocated
        // don't release mutual exclusion lock and immediately requests second stick
        while ( !available [(me + 1)% N]) {
            try { wait() ; } catch (InterruptedException e) {}
        }
        available [(me + 1)% N] = false ; score = 2; // both sticks allocated
    }

    public synchronized void release (int me) {
        available [me] = true ; available [(me + 1)% N] = true ;
        notifyAll() ;
    }
}
```

3.2. Ada transcription of the Java monitor policy

The transcription has to cope with two main differences between Java and Ada: the localisation of the blocking statements into the code and the awaking policy.

1. In Java, wait() is a method of the Object Class and one or several wait() calls may appear in the code; thus a signalled thread exiting from the wait() method returns inside the code at the instruction immediately following the wait() call. In Ada, a task calling an entry waits if the barrier of this entry is false and there is no waiting possibility inside the entry code. Thus a signalled task must always start at the beginning of the entry code. In the transcription into Ada, the Java get_LR() method code which contains two calls to wait() is sliced in three sequences of code delimited by these calls and referenced using a pseudo program counter, named score, and the Ada entry, Get_LR(), uses a case statement for selecting the code alternative to execute when the calling task is signalled.
2. This Java implementation style is simulated by an Ada program mimicking the Java monitor behaviour. The Java method is represented by the procedure Get_LR() calling the Sticks.Get_LR() entry as long as the allocation is not completed. Sticks.Get_LR() entry barrier is always True. An entry Sticks.Wait() provides the unique waiting queue for condition synchronization. Once notified, all tasks queued at Sticks.Wait() leave the protected object, none are requeued, and the Get_LR() procedure calls again Sticks.Get_LR(), competing anew for the monitor lock. This is repeated until the calling task gets all its sticks allocated.

```
generic
    Type Id is mod < >;
    -- instanciated as mod N
package Chop is
    procedure Get_LR(C : Id);
    procedure Release(C : Id);
end Chop;


package body Chop is
    type SticState is array(Id) of Boolean;
    type Cardinal is new Integer range 0..2;

    protected Sticks is
        entry Get_LR(C : Id; Score : in out Cardinal);
        -- Score is the number of already allocated sticks
        procedure Release(C : in Id);
    private
        entry Wait(C : Id; Score : in out Cardinal);
        Available: SticState := (others => True);
        NotifyAll : Boolean := False; -- Java signal simulation
    end Sticks;
```

```
    protected body Sticks is

        entry Get_LR(C : Id; Score : in out Cardinal) when True is
        begin
            case Score is
                when 0 =>    if Available(C) then
                                 Available(C) := False; Score:= 1; -- Left stick allocated
                                 if Available(C + 1) then
                                     Available(C + 1) := False; Score := 2;
                                     -- Right stick also allocated on the spot
                                 end if;
                             end if;
                when 1 =>    if Available(C + 1) then
                                 Available(C + 1) := False; Score := 2;
                                 -- Right stick now also allocated;
                             end if;
                when 2 =>    null;
            end case;
            if Score /= 2 then
                NotifyAll := False; requeue Wait; -- Java wait simulation
            end if;
        end Get_LR;

        entry Wait(C : Id; Score : in out Cardinal) when NotifyAll is
        begin          -- when NotifyAll is True, the eggshell model gives precedence to all queued tasks
            null;      -- all queued tasks quit the monitor
        end Wait;

        procedure Release(C : Id) is
        begin
            Available(C) := True; Available(C + 1) := True;
            NotifyAll := (Wait'Count > 0); -- Java NotyfyAll simulation
        end Release;

    end Sticks;

    procedure Get_LR(C : Id) is
        Score : Cardinal := 0;
    begin
        -- the following participate simulating the Java signalling semantics
        while Score /= 2 loop
            -- possibly overtaken by another philosopher while in the system ready queue
            Sticks.Get_LR(C, Score);
        end loop;
    end Get_LR;

    procedure Release(C : Id) is
    begin Sticks.Release(C); end Release;

end Chop;
```

These two preceding programs (the Java one as well as the Ada one transcribing it) usually run correctly and this may give false confidence. However these programs are not safe. They occasionally fail and deadlock, but this is a situation which is rare, difficult to reproduce and therefore to explain and debug.

3.3. Ada regular implementation

In order to get insight, the same program is implemented letting the Ada monitor behave normally, i.e. respecting the eggshell semantic. Once notified, a waiting task does not leave the protected object and is requeued to Get_LR(). It has precedence over new requestors. If several tasks are queued at the wait() entry, all of them must be requeued prior letting one of them execute the Get_LR() entry call.

```
package body Chop is
    type SticState is array(Id) of Boolean;
    type Cardinal is new Integer range 0..2;
    type SticScore is array(Id) of Cardinal;

    protected Sticks is
        entry Get_LR(C : Id);
        procedure Release(C : in Id);
    private
        entry Wait(C : Id);
        Available : SticState := (others => True); -- Stick availability
        Score : SticScore := (others => 0); -- allocation state
        StillToNotify : Integer := 0;
        NotifyAll : Boolean := False; -- Java signal simulation
    end Sticks;

    protected body Sticks is

        entry Get_LR(C : Id) when not NotifyAll is
            -- after each execution of Release
            -- the Wait entry queue must be fully emptied before serving Get_LR anew
        begin
            case Score(C) is
                when 0 =>   if Available(C) then
                                Available(C) := False; Score(C):= 1; -- Left stick allocated
                                if Available(C + 1) then
                                    Available(C + 1) := False; Score(C):= 2;
                                    -- Right stick also allocated on the spot
                                end if;
                            end if;
                when 1 =>   if Available(C + 1) then
                                Available(C + 1) := False; Score(C) := 2;
                                -- Right stick now allocated;
                            end if;
                when 2 =>   null;
            end case;
            if Score(C) /= 2 then
                requeue Wait; -- Java wait simulation
            end if;
        end Get_LR;

        entry Wait(C : Id) when NotifyAll is
        begin
            StillToNotify := StillToNotify - 1;
            NotifyAll:= StillToNotify > 0; -- when False, opens the Get_LR barrier
            requeue Get_LR; -- all signalled tasks remain inside the monitor
        end Wait;
```

```
            procedure Release(C : Id) is
            begin
                Available(C) := True; Available(C + 1) := True; Score(C) := 0;
                -- awakes all blocked tasks as with Java's NotifyAll
                StillToNotify := Wait'Count;
                NotifyAll := StillToNotify > 0;
            end Release;
        end Sticks;

        procedure Get_LR(C : Id) is begin Sticks.Get_LR(C); end Get_LR;
        procedure Release(C : Id) is begin Sticks.Release(C); end Release;
    end Chop;
```

This implementation is reliable, fair and never deadlocks when running. Quasar, our tool for concurrent Ada analysis [Evangelista 2003], has validated its correctness and has given also a sequence of actions that leads the Java program to a deadlock.

Let us consider the following running sequence. Philosophers request the sticks in the following sequential order: 4, 3, 2, 1, 0. Philosopher 4 takes two sticks and eats while Philosophers 3, 2 and 1, one after the other, take their left stick and wait for their right stick that they find already allocated. Philosopher 0 finds that its left stick has been allocated, so it waits for it. As soon as Philosopher 4 has released its two sticks, it becomes hungry anew and calls Get_LR immediately. Suppose that Philosopher 0, which has been signalled of its stick availability, has taken its left stick in the meanwhile and now waits for its right one.

The correctness depends on the choice of the next process that will access the monitor. If it is Philosopher 3, it will take its right stick and eat. If it is Philosopher 4, it will take its left stick and find its right stick already allocated. It will be blocked, as already are the four other Philosophers, and this is a deadlock.

The Java policy allows Philosopher 4 to compete for acquiring the access lock of the object chop, and if it succeeds occasionally to take precedence over Philosopher 3, this will cause a deadlock. Ada gives always precedence to the existing waiting calls, that is Philosopher 3 has always precedence over Philosopher 4 and there is never a deadlock.

This shows that the correctness relies sometimes on the concurrency semantic of the run-time system. It shows also why deadlock is not systematic in the Java implementation, and why this non-deterministic behaviour makes its correctness difficult to detect by tests.

Deadlock is prevented if a philosopher already owning its left stick books immediately its right stick, forbidding its right neighbour to get precedence for acquiring it anew as a left stick. This leads to Java and C# defensive implementations that are reliable but not fair (see Annex 1).

4. An Ada deterministic deadlock free and fair implementation

All the preceding implementations use a unique waiting queue and this implies a semi-active awaking solution. Semi-active means that any waiting process must be signalled even if its blocking condition has not changed and if it is doomed to wait anew. Programming several condition synchronization queues allows refining the implementation of our new solution of the dining philosophers paradigm in a more attractive solution. In Ada, the use of entry families and requeue statement provides then an efficient and correct (i.e., no deadlock, no starvation) implementation with at most one task waiting at an entry, with requeuing just once (no semi-active wait while scanning entry queues) and with a simple Boolean variable as entry barrier. Moreover this solution is very elegant and deserves being presented as a tribute to Don Knuth who wrote in the preface of its Art of Computer Programming "*The process of preparing programs can be an aesthetic experience much like composing poetry or music*"[Knuth 1969].

```
package body Chop is
    type SticState is array(Id) of Boolean;

    protected Sticks is
        entry Get_LR(Id); -- entry family
        procedure Release(C : in Id);
    private
        entry Get_R(Id); -- entry family
        Available : SticState := (others => True); -- Stick availability
    end Sticks;

    protected body Sticks is

        entry Get_LR(for C in Id) when Available (C) is
        begin Available (C) := False; requeue Get_R(C + 1) ; end Get_LR;
        -- Left stick is allocated

        entry Get_R(for C in Id) when Available (C) is
        begin Available (C) := False; end Get_R;
        -- stick C is allocated as a right stick

        procedure Release(C : Id) is
        begin Available(C) := True; Available(C + 1) := True; end Release;
    end Sticks;

    procedure Get_LR(C : Id) is begin Sticks.Get_LR(C); end Get_LR;
    procedure Release(C : Id) is begin Sticks.Release(C); end Release;
end Chop;
```

This style of programming is inherited from the private semaphore [Dijkstra 1968] and the original monitor [Hoare 1974] proposals where a waiting process is awaken only when it has been granted all the resources it requested.

The waiting queues contain at most one task and thus the queuing policy has no influence. Since there is always one and only one requeue operation executed, there is no task recycling among entry queues. All these features contribute reducing non-determinacy (see section 6.1 and Annex 2). This leads us to suggest an extension of the Ravenscar profile [Burns 2004], allowing protected objects with entry families and requeue statements conjugated with the restriction of at most one task per entry and one requeue execution per task.

This implementation is safe and fair. If necessary, priorities may be given to tasks. Entries Get_R may have priority over Get_LR. This may also be stated in the program, as below.

```
entry Get_LR(for C in Id) when Available (C) and Get_R(C)'Count = 0 is
begin
    Available (C) := False; -- Left stick allocated
    requeue Get_R(C + 1) ;
end Get_LR;
```

Another interest of an implementation that has got rid of the queuing policy influence is that it is easy to trace the allocation process and to undo it when a client is aborted. Suppose that the Ada "requeue with abort" clause is added. When a task waiting at Get_LR() is aborted, it has no stick reserved yet for it and the abortion has no effect on the resource availability, while aborting a task waiting at Get_R() causes a stick leakage and ruins the problem stability. A policy for propagating abortion signals to abortion handlers, much alike the one used for exceptions, would be welcome in some future Ada language revision. It would allow releasing the already allocated stick, as programmed below.

```
entry Get_LR(for C in Id) when Available (C) is
begin
    Available (C) := False; -- Left stick allocated
    requeue Get_R(C + 1) with abort;
end Get_LR;

entry Get_R(for C in Id) when Available (C) is
begin
    Available (C) := False; -- stick allocated as a right stick
when aborted => Available (C – 1) := True ; -- abortion handler, not possible in Ada to-day
end Get_R;
```

5. Chops Global allocation

Chops Global allocation allows the largest number of jointly eating philosophers and therefore is a useful benchmark when comparing implementations. In this solution, the chopsticks are allocated globally. The use of entry families provides again a simple Ada implementation. The corresponding part of the protected object body is then:

```
entry Get_LR(for C in Id) when Available (C) and Available(C + 1) is
begin
    Available (C) := False; -- Left stick allocated
    Available (C + 1) := False; -- Right stick allocated
end Get_LR;  -- no requeue needed
```

However, this solution allows starvation when a postponed philosopher has always one of its neighbours eating. In Java, starvation may additionally occur when a releasing philosopher calls immediately Get_LR and gets anew the monitor lock before a signalled philosopher.

6. Instrumentation

6.1. Concurrency complexity

Our verification tool Quasar [Evangelista 2003] translates automatically a concurrent Ada program into a formal model. The size of the generated model can be used as a concurrency

complexity measure. Table 1 records different implementations of the dining philosophers which complexity have been thus measured:

a. Unsafe Chop Java with Java semantics simulated in Ada (section 3.2.),
b. Corrected unsafe Chop Java, with Java semantics simulated also in Ada,
c. Reliable Chop Ada with busy re-evaluation as in Java (section 3.3),
d. Reliable Chop Ada with entry families and requeue (section 4),
e. Global chop allocation in Ada (section 5),
f. Dummy protected object providing a concurrent program skeleton.

Program	ColouredPN #places	ColouredPN #transitions	Chop part of places& trans	Reachability #nodes	Reachability #arcs
a Unsafe JAVA	127	103	67 & 61	39 620	42 193
b Reliable JAVA	136	111	76 & 69	37 445	39 558
c ADA as JAVA	129	111	69 & 69	107 487	118 676
d ADA Family	96	75	36 & 33	3 860	4 244
e ADA Global	89	68	29 & 26	2 110	2 344
f Program Skeleton	60	42	0 & 0	22	22

Table 1. Complexity measures given by Quasar

Our tool Quasar first generates a coloured Petri net model of the program, which is simplified by structural reductions. Hence it takes advantage of symmetries, factorizations and hierarchies present in the program text. Thus the number of elements of the Petri net is related to programming style. Second, Quasar performs model checking, generating a reachability graph which records all possible different executions of the program: the least number of elements in the graph, the least task interleaving. The graph size is thus related to the execution indeterminacy.

The implementations d and e, which use entry families, receive good marks for style and determinacy, while, in implementations b and c, the use of a unique waiting queue with a semi-active awaking creates much more combinatorics.

6.2. Concurrency effectiveness

The different implementations have been instrumented and simulated in order to measure the number of times philosophers eat jointly, i.e. the effective concurrency. The instrumentation analyses also why a stick allocation is denied, whether it is structural, i.e., because one of the neighbours is already eating, or it is cautious, i.e. for preventing deadlock or starvation.

Table 2 records the data collected after running 100 000 requests performed by a set of five philosophers. They think and eat during a random duration, uniformly distributed between 0 and 10 milliseconds. Since the Java monitor semantics implies that a notified thread joins the ready thread queue, the Java implementation is sensitive to the number of threads contending for the monitor lock. The contention probability is enlarged by adding a delay, also uniformly distributed, between the notification and the lock request. A run (labelled 1 milli) is performed with a delay between 0 and 1 milliseconds, another (labelled 10 micro) with a delay between 0 and 10 microseconds, a third one with no delay (labelled zero). The data collected are:

PairRatio: ratio of times a philosopher starts eating while another is already eating,
SingletonsRatio: ratio of times a philosopher starts eating alone,
NbStructuralRefusals: number of denials due to a neighbour already eating,
NbCautiousRefusals: number of denials due to deadlock or starvation prevention,
Simulation time: duration of the simulation run,
Allocation time: mean allocation time for a philosopher during the simulation,
Allocation ratio: ratio of time used allocating the chopsticks.

Program 100 000 requests	PairRatio	Singleton Ratio	NbStructural Refusals	NbCautious Refusals	Simulation time (s.)	Allocation time (s.)	Allocation ratio
Reliable JAVA 1 milli	22%	78%	95 643	82 539	604	381	63%
Reliable JAVA 10 micro	36%	64%	92 177	79 837	480	254	53%
Reliable JAVA zero	36%	64%	101 595	89 638	564	309	55%
ADA as JAVA	43%	57%	100 010	86 433	512	249	49%
ADA family	42%	58%	45 318	28773	500	245	45%
ADA Global	87%	13%	68 602	0	297	85	29%

Table 2. Concurrency effectiveness given by simulations

In this simulation, the best effective concurrency, i.e., the number of times two philosophers eat jointly, is provided by the global allocation (which however allows starvation). The programs with the eggshell semantics have similar effective concurrency values, which are only 50% of the former. The programs simulating the Java monitor semantics have less effective concurrency. Augmenting the delay between the notification and the lock request, and therefore the possibility of contention, decreases the ratio of pairs. This simulation points out also that the Java programming style with a unique implicit condition queue and thus with dynamic re-evaluation of requests, whether outside or inside the monitor, involves much more denials than the deterministic entry families and requeue once Ada implementation. This observation might be correlated with the reachability graph size showing execution indeterminacy.

The analysed programs and the data collected are available on Quasar page [Quasar 2006] at: http://quasar.cnam.fr/files/concurrency_papers.html

7. Conclusion

Former experience in developing concurrent processes in operating systems and real-time applications [Bétourné 1971], as well as in teaching concurrent programming [ACCOV 2005], gives us credit to assert that Ada is the imperative language with the best set of tools for reliable concurrent programming, especially with the conjugate use of protected object, requeue and entry families. Our measurements seem to corroborate this statement.

However, due to Ada past and Ada 83 early choice of rendez-vous, the protected object power has been underestimated and its full potentiality not yet used.

We therefore recommend the following approaches for taking full advantage of its capabilities.

1. Conjugate algorithmic design and validation [Kaiser 1997] in order to find out the simplest and most elegant structures, which are also the easiest to debug and to prove correct.
2. Compare radically different implementations of a given paradigm, showing the variety of styles and concurrency efficiency of reliable concurrent programming.
3. Use a tool like Quasar to validate the correctness of solutions and to compare their concurrency complexity.

Simplicity in the design is always the result of a long cohabitation with a problem and of a better understanding of it. This is normal scientific progress, and concurrent programming should also take part of it.

Concurrent programming is still a challenge. However compared to Java and C# which basic choices require defensive multithreading programming, the Ada concurrency features associated with the protected object provide a strong basis allowing a more open, diverse and offensive approach when developing reliable concurrent programs. Note yet that the monitor implementation using an Ada server task and rendez-vous, which don't respect the eggshell semantics, suffers of the same weakness as Java and C# and also requires defensive concurrent programming. Examples are given in [Kaiser 1997].

8. References

[ACCOV 2005] http://deptinfo.cnam.fr/Enseignement/CycleSpecialisation/ACCOV/

[Ada 1999] Ada 95 Reference Manual, International Standard ANSI/ISO/IEC-8652: 1995, January 1999.

[Barkaoui 1997] K. Barkaoui, C. Kaiser and J.F. Pradat-Peyre. Petri nets based proofs of Ada95 solution for preference control. *Joint APSEC97/ICSC97 Conference.* Hong-Kong December 1997, 15 pages. IEEE CS Press 1997.

[Bétourné 1971] C. Bétourné, J. Ferrié, C. Kaiser, S. Krakowiak and J. Mossière. System Design Using Parallel Processes. *IFIP congress 1971*. pp. 345-352. North Holland. 1971.

[Brosgol 1996] B. Brosgol. The dining philosophers in Ada95. *In Reliable Software Technologies-Ada-Europe'96*, LNCS 1088, pp. 247-261. Springer-Verlag, 1996.

[Buhr 1995] P. Buhr, M. Fortier, M. Coffin. Monitor Classification, *ACM Computing Survey*, 27,1, pp. 63-107. 1995.

[Burns 1995] A. Burns and A. Wellings. *Concurrency in Ada*, Chapter 6.11, pp. 134-137. Cambridge University Press, 1995.

[Burns 2004] A. Burns, B. Dobbing and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *ACM SIGAda Ada Letters*, 24,2. pp. 1-74, June 2004.

[Dijkstra 1968] E.W. Dijkstra. The Structure of the "THE" Multiprogramming System. *Communications of the ACM*, 11,5. pp. 341-346, May 1968.

[Dijkstra 1971] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, number 1, pp. 115-138, 1971. (also available as EWD310 at http://www.cs.utexas.edu/users/EWD/)

[Evangelista 2003] S. Evangelista, C. Kaiser, J.F. Pradat-Peyre, P. Rousseau. Quasar : a new tool for analyzing concurrent programs. *International Conference on Reliable Software Technologies, Ada-Europe'03*, LNCS vol. 2655, pp. 166-181, Springer-Verlag 2003. Toulouse, France, June 2003.

[Hoare 1974] C. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*. 17,10. pp. 549-557. October 1974.

[Intermetrics 1995] Intermetrics Inc. *Ada 95 Rationale*, The Core Language, 9 Tasking, 9.1 Protected Types, page 9-5, Cambridge, Mass.,1995.

[Kaiser 1997] C. Kaiser and J.F. Pradat-Peyre. Comparing the reliability provided by tasks or protected objects for implementing a resource allocation service : a case study. *Tri-Ada'97 Conference*, Saint-Louis, USA. pp. 51-65. ACM publication, 1997.

[Knuth 1969] D. Knuth. *The Art of Computer Programming, Volume 1. Fundamental Algorithms*. 634 pages. Addison-Wesley 1969.

[Lea 1998] D. Lea. Patterns and the democratization of concurrent programming, *IIE Concurrency, Parallel, Distributed & Mobile Computing*, 6,4, pp. 11-13, 1998.
[Quasar 2006]. http://quasar.cnam.fr/

Annex 1. Defensive and therefore safe Java implementation

A defensive solution consists in giving precedence to a philosopher already owning its left stick and requesting its right stick over a philosopher requesting its left stick. A stick may be booked for being used as a right stick and forbidding its use as a left stick. This leads to a safe although unfair solution (no deadlock, possible starvation).

```
public final class Chop {

    private int N ;
    private boolean available[ ] ;
    private boolean bookedRight[ ] ;

    Chop (int N) {
        this.N = N ;
        this.available = new boolean[N] ;
        this. bookedRight = new boolean[N] ;
        for (int i =0 ; i < N ; i++) {
            available[i] = true ; // non allocated stick
            bookedRight[i] = false;
        }
    }

    public synchronized void get_LR (int me) {
        while ( !available[me] || bookedRight[me]) {
            try { wait() ; } catch (InterruptedException e) {}
        }
        available[me] = false ; // left stick allocated
        bookedRight[(me + 1)% N] = true; // right stick booked

        // don't release mutual exclusion lock and immediately requests second stick
        while ( !available[(me + 1)% N]) {
            try { wait() ; } catch (InterruptedException e) {}
        }
        available[(me + 1)% N] = false ; // both sticks allocated
        bookedRight[(me + 1)% N] = false; // no more reason for booking right stick
    }

    public synchronized void release (int me) {
        available[me] = true ; available[(me + 1)% N] = true ;
        notifyAll() ;
    }
}
```

Annex 2. Waiting times with Ada deterministic deadlock free and fair implementation

Let us imagine a real-time application related to the dining philosophers paradigm. Tasks collect information from some instruments, process it and prepare control orders for external devices and reports for remote data distribution. Connexion to the devices requires links to shared resource and reports transmission requires some shared data channels availability. Suppose that these tasks are periodically triggered with a minimum inter-arrival time of T and that the question is to estimate the worst response time of each task, and to fix the minimum value of T that will prevent overloading the processor. A philosopher i is defined by:

- a worst case execution time Ci = Thi + Eai
  where Thi is its longest thinking time and Eai is its longest eating time,
- a worst case blocking time Bi introduced by shared resource contention.

Thus its worst response time Ri is Ri = Ci + Bi.

The worst response time corresponds to an execution state where a maximum number of philosophers are waiting and have precedence over philosopher i. This is the situation that may precede deadlock in Java and that has been analysed in section 3.3. above.

Let us consider again the running sequence which corresponds to philosophers requesting the sticks in the following sequential order: 4, 3, 2, 1, 0.

Philosopher 4 takes two sticks and eats while Philosophers 3, 2 and 1, one after the other, take their left stick and wait in the Get_R() queue for their right stick that they found yet allocated. Philosopher 0 finds its left stick already taken, so it waits for it in the Get_LR(0) queue. When philosopher 4 releases its two sticks, both of its neighbours become eligible. Philosopher 0 gets its left stick and now waits in Get_R(1) queue for its right one. Philosopher 3 takes its right stick and eats. As the inter-arrival rate T is greater than the waiting philosophers response time, philosopher 4 will not request the sticks in the meanwhile. When Philosopher 3 releases its two sticks, philosopher 2 can receive its right stick and eat. And so on.

Whatever the underlying processor scheduling policy, the sole possible philosophers execution sequence is 4, 3, 2, 1, 0. This running situation leads to a FIFO service whatever are the philosopher's priorities. Hence the last of the original sequence, philosopher 0, will eat before any other one can eat a second time.

The run-time kernel is activated by philosopher 0 calling Get_LR(0), and then by each philosopher releasing its sticks. Each of these activations have to evaluate or re-evaluate 2 barriers and execute two entry bodies. Recall that the Ada 95 Rationale [Intermetrics 1995] indicates that these run-time executions can be done by the releasing tasks minimizing unnecessary context switches. The static analysis of the kernel worst execution time can be done easily. The worst case waiting time of Philosopher 0 is therefore the sum of :

- the 4 other philosophers eating time: Ea1+ Ea2 + Ea3 + Ea4,
- plus 5 times the kernel worst execution time: 5*K,
- plus 5 task context switches: 5*S.

The first part is specific to the chosen algorithm while the others are implementation dependant.

With N philosophers triggered with a minimum inter-arrival time of T, greater than the lowest priority philosopher response time, the worst response time of the highest priority philosopher is:

$$R0 = Th0 + SIGMA\ \{Eaj\ (for\ j\ in\ 0..N-1)\} + (N)*(K + S)$$