Weak fairness semantic drawbacks in Java multithreading

Reliable Software Technologies
Ada – Europe 2009

Claude Kaiser[1], Jean-François Pradat-Peyre[2]

[1] CEDRIC - CNAM Paris
292, rue St Martin, F-75003 Paris, France
{claude[dot]kaiser}[at]cnam[dot]fr
[2] LIP6 - Université Pierre et Marie Curie
104 avenue du Président Kennedy, F-75016 Paris, France
{Jean-Francois[dot]Pradat-Peyre}[at]lip6[dot]fr
http://quasar.cnam.fr/

Abstract. With the development of embedded and mobile systems, Java is widely used for application programs and is also considered for implementing systems kernel or application platforms. It is the aim of this paper to exemplify some subtle programming errors that may result from the process queuing and awaking policy, which corresponds to a weak fairness semantic and which has been chosen for implementing the monitor concept in this language.

Two examples show some subtle deadlocks resulting from this policy. The first example deals with process synchronization: processes seeking after partners for a peer-to-peer communication call a symmetrical rendezvous server. The second example concerns resource sharing according to a solution of the dining philosophers paradigm. In this example, several implementations are presented, the last ones aiming to provide deterministic process awakening. All these examples have been validated and simulated and this allows comparing their concurrency complexity and effectiveness.

Our conclusion is, first, that the use of Java for multithreading programming necessitates sometimes additional shielding code for developing correct programs and, second, that a good acquaintance with several styles of concurrent programming helps designing more robust Java solutions, once the choice of the implementation language is irrevocable.

1. Introduction

1.1. Concurrency programming

Java is widely used for application programs and, with the development of embedded and mobile systems, it is also considered for implementing systems kernel or application platforms. .

Concurrent programming is a prolific source of complexity. Thus it is a serious cause of errors when developing applications, system kernels or platforms. One proper engineering solution is to choose a good level of abstraction for concurrency control. For this reason the monitor concept [Hoare 1974] has been implemented in past operating systems, as early as in the personal computer system Pilot with the Mesa language [Redell 1980, Lampson 1980].

Java designers are aware that concurrent programming is difficult and is still a challenge for developers. *"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 1998]. "*Providing significant examples and paradigms is of prime importance for mastering good and correct style. Even if you never employ them directly, reading about different special-purpose design patterns can give you ideas about how to attack real problems*" [Lea 1997].

When implementing multithreading, choosing reliable concurrent algorithms is necessary, however it is not sufficient. The behavioural context must also be considered since subtle running errors often arise from the semantics of the run time kernel or the underlying platform. It is the aim of this paper to point out some possibly negative consequences of the concurrency semantic chosen for the Java language.

1.2. Overview of the monitor concept as implemented in Java

Several possible monitor concurrency semantics have been used in the past and a classification has been presented in [Buhr1995]. Every implementation has to provide mutual exclusion during the execution of a distinguished sequence . However an implementation may have a specific policy for blocking, signalling and awaking processes. The languages Java and C# both include the monitor concept and have chosen the same run time policy.

The Java language provides mutual exclusion through synchronized block or synchronized method, using a lock for every object, and 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 awaked 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 awaked thread (the lock availability is globally visible), another thread contending for the monitor may take precedence over awaked threads that have already been blocked on the waiting queue. This awaking policy involves weak fairness. If this elected thread calls also a synchronized method (or enters a synchronized block) of the object, it will acquire the lock before the awaked threads and then access the object before them. This may contravene some problem specifications and in that case may require adding some shielding code to maintain the original algorithmic correctness.

Since Java 1.5, the basic Java monitor has been extended and allows using multiple named condition objects. This provides more programming flexibility, however the signalling policy remains the same and the weak fairness semantic is still present.

The language C# has also thread synchronization classes using for example Wait(), Pulse(), Monitor.Enter(), Monitor.Quit(). Its thread queuing and signalling policy relies also on a weak fairness semantic. Thus it has the same drawbacks as Java.

1.3. Outline of the paper

Process synchronization and resource sharing are basic concerns when developing concurrent software. We present an example in each domain. Each will show some subtle consequences of Java's basic policy that may lead to deadlock. The first example is a symmetrical rendezvous server, called by processes seeking after partners for a peer-to-peer communication. The second example concerns resource sharing and is a new solution of the

dining philosophers paradigm. Several implementations are presented, using different semantics choices, the Java one and another one that gives precedence to the awaked processes, i.e. that implements strong fairness. It will be shown that in this case, which is the Ada choice, the implementations are simpler and safer.

All these examples have been submitted to our verification tool Quasar and have also been simulated for performance comparison. This allows comparing their concurrency complexity and effectiveness.

Our conclusion is, first, that the use of Java for multithreading programming may necessitate additional shielding code when implementing concurrency algorithms which have been proven correct in strong fairness context and, second, that good acquaintance with several styles of concurrent programming is helpful for designing better Java solutions for concurrent applications, once the choice of the implementation language is irrevocable.

General appraisals of the Java concurrency features as well as their comparison with Ada have been published [Brosgol 1998, Potratz 2003]. Our paper focuses on the incidence of fairness semantic on reliability and our appraisal is supported by a concurrency verification tool.

2. Process synchronization example: symmetrical rendezvous paradigm

The synchronization example is the mutating chameneos paradigm [Kaiser 2003]. It involves a symmetrical rendezvous before peer-to-peer cooperation of concurrent processes. The cooperation, depicted as a possible colour mutation between the chameneos of a pair, is not developed in this paper. Here we cope only with a solution where a chameneos eager to cooperate calls a rendezvous server in order to find a partner.

The rendezvous server has the following specification:

1- the server must wait until it has received two requests before giving notification,
2- multiple requests shall not disturb the service,
3- notifications must be sent as soon as possible,
4- once A and B are paired, A must know that its partner is B and B that its partner is A.

A possible server behaviour, respecting mutual exclusion, is:

- at the first call, the server registers the name of the first caller; then it waits the end of second call before reading the name of the mate and returning it to the first caller.

- at the second call, it registers the name of the second caller, reads the name of the mate, notifies it to the second caller, and wakes the first request, signalling that its mate name is now available.

This algorithm has been proven reliable by our tool Quasar when implemented in Ada (i.e. with a strong fairness semantic).

The corresponding Java class is given in Program 1. A synchronized method is used.

```
public class Rendez_Vous {
    private ThreadId APartner, BPartner;      // names of the first and second requesting thread
    private boolean FirstCall = true;         // false when SecondCall
    private boolean MustWait = false;         // used for defensive code

    public synchronized ThreadId partner(ThreadId x){
        ThreadId result;

        // the following loop is a necessary defense, forbidding access by a third partner
        while (MustWait){
            try{wait();} catch(InterruptedException e){}
```

```
        }

        // the following is the code solving the specifications of the problem
        if (FirstCall){
            APartner = x; FirstCall = false;    // now the caller must wait the end of the second request
            while ( !FirstCall ){
                try{wait();} catch(InterruptedException e){}
            }
            MustWait = false; result = BPartner; notifyAll();
        }
        else{
            BPartner = x; result = APartner; FirstCall = true; MustWait = true; notifyAll();
        }
        return result;
    }
}
```

Program 1. Symmetrical rendezvous implementation in Java

Due to the Java choice of locking and notifying semantics, if the access of a third chameneos had not been explicitly forbidden in the code (this is done using MustWait), the program would have been erroneous. Indeed, suppose that the barrier with MustWait is not implemented and consider four requests, A, B, C and D: partner(A) may be D, partner(B) may be A, partner(C) may be D and partner(D) may be C. A and B are not correctly paired and this leads to deadlock.

A semantic choice giving precedence to the awaked processes, as in Ada, removes the need of this defensive barrier. Java, Ada and Posix solutions are compared in [Kaiser 2003].

This simple program is our first example showing that the Java implementation of a correct algorithm must care of the underlying semantics and that this care may often lead to add compulsory shielding code.

3. Resource allocation example

3.1. Another solution of the dining philosophers paradigm

The dining philosophers, originally posed by Dijkstra [Dijkstra 1971], are 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: deadlock and starvation.

This paradigm has two well-known approaches: step-wise or global allocation [Hartley 1998, ACCOV 2005]. Let us consider now another approach, which has been experimented in [Kaiser 1997] and which has been proven correct by our tool Quasar when implemented with a strong fairness semantic (in Ada). The chopsticks are allocated as many as available and the allocation is completed as soon as the missing chopsticks are released.

3.2. Straightforward Java implementation

A straightforward Java implementation of this latter solution leads to the following Chop class with get_LR and release methods (Program 2). It is influenced by the Java choice of a monitor with a unique and implicit condition queue.

```
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) {
        while ( !available[me]) {
            try { wait() ; } catch (InterruptedException e) {}
        }
        available[me] = false ; // 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 ; // both sticks allocated now
    }

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

Program 2. Unsafe Chop implementation in Java

This Java implementation may give misleading confidence. Actually the program is not safe. It occasionally fails and deadlocks, but this is a situation which is rare, difficult to reproduce and therefore to explain and debug.

3.3. Deadlock analysis

Let us consider the following running sequence. Philosophers request the sticks in the following sequential order: 4, 3, 2, 1, and 0. Philosopher 4 takes two sticks (sticks 4 and 0) 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 taken, 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 has been awaked first and in the meanwhile has taken its left stick and now waits for its right one (stick 1). The correctness relies on the choice of the next process that will access the monitor and take stick 4. 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. If precedence were given to the existing waiting calls (as it is the semantic choice of Ada 95 and also of the private semaphore schema [Dijkstra 1968]), Philosopher 3 would have always precedence over Philosopher 4 and there would never be 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 solution, and why this non-deterministic behaviour makes its correctness difficult to detect by tests.

3.4. Defensive and therefore safe Java straightforward implementation

A safe solution is achieved 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 must be booked for being used as a right stick and forbidding its use as a left stick (this is programmed using the boolean bookedRight[ ] as a barrier). This leads to a safe although unfair solution (no deadlock, possible starvation). It is given below as a subset of the safe and fair solution.

This approach needs shielding code because of the Java monitor policy. Care had to be taken of the underlying platform behaviour, as quoted, in java.lang.Object, in the wait method detail section. *"A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup... In other words, waits should always occur in loops."*

This solution is safe but unfair. For example suppose philosopher 4 eats (it uses sticks 4 and 0) while philosopher 0 is hungry and is waiting for stick 0. Suppose now that after releasing its sticks, philosopher 4 requests them immediately, calling get_LR. Philosopher 4 may acquire the monitor lock anew before philosopher 0, involving starvation of the latter.

A fairness constraint may be derived from this example, prescribing that a releasing philosopher cannot get again a stick as its right stick before an already waiting philosopher, which has previously booked this stick as its left stick (the boolean bookedLeft[ ] is used as a barrier for programming it). However this fairness constraint is circular and re-introduces deadlock, unless a releasing philosopher is denied a new access to the monitor when one of its neighbours is already waiting for one of its released sticks. This gives precedence to already waiting philosophers. Additional shielding code is again necessary because of the chosen monitor policy. Whence Program 3 which is a solution proven correct by our verification tool Quasar:

```
public final class Chop {

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

    Chop (int N) {
        this.N = N ;
        this.available = new boolean[N] ;        // chop availability when true
        this. bookedRight = new boolean[N] ;  // always compulsory for deadlock avoidance barrier
        this.bookedLeft = new boolean[N] ;     // used only when fairness is required
        for (int i =0 ; i < N ; i++) {
            available[i] = true ; bookedRight[i] = false; bookedLeft[i] = false;
        }
    }

    public synchronized void get_LR (int me) {
        // compulsary defensive code giving precedence to already waiting philosophers when seeking fairness
        while ( bookedRight[me]|| bookedLeft[me + 1)% N){
            try { wait() ; } catch (InterruptedException e) {} // a stick has been booked by one neighbour
        }
        while ( !available[me] || bookedRight[me]) { // deadlock avoidance barrier even when unfair allocation
            try { bookedLeft[me] = true ; wait() ; } catch (InterruptedException e) {}
            // bookedLeft[me] reserves left stick if fairness is requested, otherwise bookedLeft is not used
        }
        available[me] = false ; // left stick allocated
        bookedLeft[me] = false; // no more reason for booking left stick
        bookedRight[(me + 1)% N] = true; // compulsary booking of right stick for deadlock avoidance
        // don't release mutual exclusion lock and immediately requests second stick
        while ( !available[(me + 1)% N] || bookedLeft[(me + 1)% N] {
```

```
            try { wait() ; } catch (InterruptedException e) {}
        }
        available[(me + 1)% N] = false ; // both sticks allocated now
        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() ;
    }
}
```

Program 3. Safe and fair Chop implementation in Java

3.5. A single waiting queue with priority given to signalled threads

This solution, with a single condition queue, has been implemented in Ada where protected objects implement a monitor where signaled threads (named tasks in Ada) take precedence over new calls. Ada implements a strong fairness semantic. This allows comparing both fairness semantics, the weak one and the strong one. The results are given in Section 5.

4. Implementations with more waiting queues

4.1. A waiting queue for each blocking condition

In the preceding solutions, all signalled philosophers have to check the availability of their sticks, even when the released sticks don't concern them. This unfortunate and inefficient behaviour is frequent in the Java programming style and it occurs naturally since a Java monitor has just one anonymous condition variable. The optimization, consisting in notifying a thread only when its both sticks have been allotted to it, provides also a more deterministic solution since it is independent of queuing policies.

Two approaches are presented in Java, the first one reproducing the private semaphore schema, the second reproducing a monitor with named condition variables. In both approaches, Java still imposes to add some defences against weakness, low-level like, with ad hoc code. These approaches have been proved safe and fair by our verification tool Quasar.

A third approach is given in Ada and allows a comparison with the semantics where priority is given to signalled threads (i.e. strong fairness). The results are given in Section 5.

4.2. A first implementation using notification objects

This solution takes inspiration from the private semaphore schema [Dijkstra 1968] and a similar one can be found in [Hartley 1998]. Mutual exclusion is provided by synchronized methods; condition synchronisation is provided by additional notification objects. A waiting philosopher is notified only when it has been allocated its requested forks, giving precedence to it over a new request of the releasing philosopher and avoiding thus a deadlock situation. It results in Program 4.

```
public final class Chop {

    private int N ;
    private boolean available[ ] ;
    private boolean requesting[ ] ; //
    private Object Allocated[ ] ; // similar role as private semaphores
    private int score[ ] ; // number of allocated sticks to each philosopher

    Chop (int N) {
        this.N = N ;
        this.available[ ] = new boolean[N] ;
        this.requesting[ ] = new boolean[N] ;
```

```
        this.Allocated[ ] = new Object[N];
        this.score = new int[N] ;
        for (int i =0 ; i < N ; i++) {
            available[i] = true; requesting[i] = false; Allocated[i] = new Object(); score[i] = 0 ;
        }
    }

    public void get_LR(int me) {
        synchronized (Allocated[me]) {
            if (successFirstTime(me)) return;        // checks condition only, no blocking in critical section
            else
                while (requesting[me]) {
                    try { Allocated[me].wait() ; } catch (InterruptedException e) {}
                    // when Allocated[me].notified will be posted,  requesting[me] will be false
                }
        }
    }

    private synchronized boolean successFirstTime (int me) {
        // successFirstTime is true when both sticks are granted; there is no blocking when it is false
        // score and requesting provide more about philosopher state
        score[me] = 0 ; requesting[me] = true ;
        if ( available[me] && available[(me + 1)% N] ){
            score[me] = 2 ; requesting[me] = false ; available[me] = false ; available[(me + 1)% N] = false ;
        }
        else if (available[me]){
            score[me] = 1 ; available[me] = false ;
        }
        return score[me] == 2 ;
    }

    public synchronized void release (int me) {
        available[me] = true ; available[(me + 1)% N] = true ; score[me] = 0 ;
        // now waiting neighbours are served preferentially and during this synchronized (critical) section
        if ( requesting[(N + me – 1)% N] && score[(N + me – 1)% N] == 1 ) {
            // the left neighbour has already its left stick and waits for its right stick since its score is one
            available [me] = false ; score[(N + me – 1)% N] = 2 ; requesting[(N + me – 1)% N] = false ;
            synchronized (Allocated[(N + me – 1)% N]) {
                Allocated[(N + me – 1)% N].notify();
            }
        }
        if ( requesting[(me +1)% N] ) {
            // the right neighbour is waiting for its first stick and its right stick is allocated also if it's available
            available[(me + 1)% N] = false ; score[(me + 1)% N] = 1 ;
            if (available[(me + 2)% N]) {
                available [(me + 2)% N] = false ; score[(me + 1)% N] = 2 ; requesting[(me + 1)% N] = false ;
                synchronized (Allocated[(me + 1)% N]) {
                    Allocated[(me + 1)% N].notify();
                }
            }
        }
    }
}
```

Program 4. Safe and fair Chop Java implementation using notification objects

The synchronized methods of Chop provide only mutually exclusive access to shared data and don't block a calling thread. The blocking test and the resulting blocking action are done in the critical section defined by synchronized(Allocated[ ]), avoiding the race conditions that may occur when a thread is notified while it has been CPU pre-empted just before calling the wait() method. This copies the private semaphore schema where shared data are accessed in

mutual exclusion without blocking and where semaphores realize an atomic test and block action.

This code is abstruse and is not easy to generalize since a very simple change may introduce deadlock. This occurs if get_LR() code is modified for avoiding embedded critical sections, and hence starts calling successFirstTime() before defining a critical section only when it returns false. A two level implementation, defining first a Java class of semaphores and using it then for implementing a private semaphore schema would result in a program using a unique synchronisation mechanism, therefore much easier to understand and less error prone.

4.3. A second implementation using JSR 166 locking utilities for JDK 1.5

J2SE/JDK 1.5 proposes Lock implementations that provide more extensive locking operations than can be obtained using synchronized methods and statements. These implementations support multiple associated Condition objects and provide an ad hoc solution to mitigate the absence of named conditions in the basic Java monitor. Where a Lock replaces the use of synchronized methods and statements, a Condition replaces the use of the Object monitor methods (http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/locks/package-summary.html).

```
import java.util.concurrent.locks.Condition ;
import java.util.concurrent.locks.Lock ;
import java.util.concurrent.locks.ReentrantLock ;

public final class Chop {
    private int N ; private boolean available[ ] ; private boolean bookedRight[ ] ;
    final Lock lock = new ReentrantLock();
    final Condition LeftAllocated[ ] ; final Condition RightAllocated[ ] ;

    Chop (int N) {
        this.N = N ;
        this.available[ ] = new boolean[N] ; this.bookedRight[ ] = new boolean[N] ;
        LeftAllocated = new Condition[N] ;
        RightAllocated = new Condition[N] ;
        for (int i =0 ; i < N ; i++) {
            available[i] = true ; bookedRight[i] = false;
            LeftAllocated[i] = lock.newCondition(); RightAllocated[i] = lock.newCondition();
        }
    }

    public void get_LR(int me) throws InterruptedException {
        lock.lock();                    // mutual exclusion entrance
        try {
            while ( !available[me] || bookedRight[me] )
                LeftAllocated[me].await();
            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] )
                RightAllocated[me + 1)% N].await();
            available[(me + 1)% N] = false ;      // both sticks allocated now
            bookedRight[(me + 1)% N] = false; // no more reason for booking right stick
        } finally {
            lock.unlock();          // mutual exclusion leave
        }
    }

    public synchronized void release (int me) throws InterruptedException {
        lock.lock();                    // mutual exclusion entrance
```

```
        available[me] = true ; available[(me + 1)% N] = true ; // waiting neighbours are served preferentially
        if (bookedRight[me]) RightAllocated[(me + N- 1)% N].signal();  // booked by the left neighbour
        LeftAllocated[(me + 1)% N].signal(); // signal the right neighbour
        // if the right neighbour is not waiting, the signal is lost
        lock.unlock();                  // mutual exclusion leave
    }
}
```

Program 5. Safe and fair Chop Java implementation using conditions and locks

In Program 5, this style improvement, due to named conditions use, allows a code that is easier to understand. It needs nevertheless the addition of booked[] variables. Moreover, even with one queue per condition, signalled threads have to request the lock again, causing more context switching and additional execution complexity.

The use of Java 1.5 extensions in an operating system kernel would suffer other criticisms. First the monitor schema with several named conditions has still to be built with low level tools: locks and queues. Second, allowing a mix of synchronization mechanisms (locks, conditions, semaphores) may lead to complicated code, difficult to debug. Accumulating and mixing synchronization concepts, although it seems cute,is not a good engineering practice, since it usually leads to code that is hard to maintain.

4.4. A monitor semantics giving priority to signalled threads

The preceding laborious implementations should be compared to the elegant simplicity of the Ada protected object and entry families solution [Kaiser 1997]. The entry family allows a set of condition queues while the requeue statement redirects the call to another condition check. This is displayed in Program 6.

```
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;

    protected Sticks is
        entry Get_LR(Id); -- entry family allowing a set of N waiting queues
        procedure Release(C : in Id);
    private
        entry Get_R(Id); -- entry family allowing a set of N waiting queues
        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;
```

Program 6. Safe and fair Chop implementation in Ada

5. Instrumentation and appraisal

5.1. Concurrency complexity

The resource allocation programs have been analysed with Quasar, our verification tool for Ada concurrent programs [Evangelista 2003]. First, Quasar generates a coloured Petri Net model of the program, which is simplified by structural reductions, taking advantage of behavioural equivalences and factorizations. Thus the size of the Petri Net (PN places and transitions) is related to programming style. Second, Quasar performs model checking, generating a reachability graph which records all possible executions of the program. Thus the least number of elements in the graph, the least combinatorics due to concurrency in the program. The graph size (reachability nodes and arcs) is related to the execution indeterminacy.

For being able to use Quasar, the Java programs have been simulated in Ada, reproducing the Java monitor semantics. This Ada transcription of the Java concurrency policy has been presented in *Ada Letters* [Evangelista 2006]. It allows checking the weak fairness semantic when strong fairness is the underlying rule.

Table 1 records the different concurrent implementations whose complexity have been thus measured:

a. Unsafe Chop Java with Java semantics simulated in Ada (Program 2.),
b. Reliable Chop Java with Java semantics simulated in Ada (see Section 3.4.),
c. Reliable and fair Chop Java with Java semantics simulated in Ada (Program 3),
d. Reliable and fair Chop Ada with a single waiting queue (see Section3.5),
e. Notification objects with Java semantics simulated in Ada (Program 4),
f. Conditions and Lock with Java semantics simulated in Ada (Program 5),
g. Conditions queues and requeue with Ada semantics (Program 6),
h. Global chop allocation also given as a useful benchmark (although not a fair solution),
i. Dummy protected object providing a concurrent program skeleton.

Program	Coloured PN #places	Coloured PN #trans	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 Fair Java	147	124	87 & 82	141 465	148 968
d Ada single	129	111	69 & 69	107 487	118 676
e Notification	170	148	110 & 106	180 585	186 708
f Conditions	158	132	98 & 90	920 924	958 931
g Ada families	96	75	36 & 33	3 860	4 244
h Global Java	116	89	56 & 47	4 745	5 073
i Skeleton	60	42	0 & 0	22	22

Table 1. Complexity measures given by Quasar

These data allow comparing the different implementations of an algorithm.

Comparing the sizes of the Petri nets provides insight on implementation simplicity and readability. The Petri net of the Java reliable and fair solution (Program 3), which is the smallest net of Java correct implementations, is one and half larger than the net generated for the Ada families implementation (Program 6).

Comparing the sizes of the reachability graph allows comparing the indeterminacy of the different implementations. The smallest graph generated for a correct Java implementation

(Program 3) is 35 times larger than the graph of the Ada families implementation (Program 6). This is partly the cost of using a weak fairness semantic and partly the result of the skilful design of the Ada protected object.

5.2. Concurrency effectiveness

The different Java implementations of the dining philosophers have been 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. The implementations have been instrumented to program explicitly the guards evaluation and to record the denial events.

Table 2 records the data collected after runs of 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. The data collected are:

NbPairs: ratio of times a philosopher starts eating while another is already eating,
NbRequestSingletons: 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	NbPairs	NbRequest Singletons	NbStructural Refusals	NbCautious Refusals	Simulation time(s)	Allocation time (s)	Allocation ratio
b Reliable Java	40%	60%	100 630	88 385	430	217	51%
c Reliable Fair Java	36%	64%	103 130	92 903	527	284	54%
d Ada single queue	43%	57%	100 010	86 433	512	249	49%
e Notification	40%	60%	31 940	41 761	550	287	52%
f Conditions	40%	60%	45 109	28 019	437	218	50%
g Ada families	42%	58%	45 318	28 773	500	245	45%
h Global Java	84%	16%	69 240	0	442	159	36%

Table 2. Concurrency effectiveness given by simulations

NbPairs, the ratio of times a philosopher starts eating while another is already eating, gives an idea of execution concurrency. The number of refusals gives an idea of the additional complexity due to condition testing dynamically.

5.3. Concurrency appraisal

These simulations show that all the different implementations of our new solution of the dining philosophers paradigm are close in effective concurrency. With the same simulation parameters, the global allocation solution doubles the number of times two philosophers eat jointly (however this solution, which is used as a benchmark for concurrency, allows starvation). The implementations with a unique condition queue and thus with dynamic re-evaluation of requests are very comparable in style, indeterminacy and number of condition denials. The solutions with several named conditions have much less refusals. The Ada family solution shows the best style measure and the least indeterminacy. The analysed programs and the data collected are available on Quasar page [Quasar 2008] at:
http://quasar.cnam.fr/files/concurrency_papers.html

6. Conclusion : Java concurrency must be handled with care

Concurrent programming is difficult and Java multithreading may reveal some misleading surprises, due to its weak fairness semantic. Using basic Java for concurrent programming implementation is a risky challenge since one cannot venture to ignore whether a concurrency paradigm remains correct or not when running in a weak fairness context. If an algorithm is fairness sensitive, a usually correct algorithm may fail and it must be reconsidered; some defence against weakness must be programmed. This additional coding is rarely obvious.

Adopting rather a strong fairness policy could reduce this risk and it could be a good choice for a future Java revision. Note that programs running safely with the current Java monitor implementation would remain safe. This is upwards compatibility. “Who who can do more can do less”. Then the shielding code would become superfluous.

More generally, our former experience in developing operating systems and real-time applications [Bétourné 1971] and also our long experience in teaching concurrency [ACCOV 2005], allows us to assert that the acquaintance with several styles of concurrent programming is helpful for choosing the best concurrency skill of each language or mechanism and then designing better Java (or C#) solutions. Better means for us simpler, safer and also more robust when well disposed, but not always fortunate, slight variations are programmed in well-known paradigms.

7. References

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

[Ada 2006] S. Tucker Taft, R.A. Duff, R.L. Brukardt, E. Plöderer, P. Leroy. *Ada 2005 Reference Manual*. XXII, 765 pages, LNCS 4348, 2006, (ISBN 978-3-540-69335-2)

[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, Benjamin M. “A Comparison of the Concurrency and Real-Time Features of Ada 95 and Java.” *SIGAda ’98 Proceedings*. ACM, pp. 175-192, 1998.

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

[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.

[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.

[Evangelista 2006] Sami Evangelista, Claude Kaiser, Jean François Pradat-Peyre, and Pierre Rousseau. Comparing Java, C# and Ada monitors queuing policies : a case study and its Ada refinement. *Ada Letters*, 26(2). pp. 23-37, ACM Press, August 2006. (ISSN:1094-3641)

[Hartley 1998] S. Hartley. *Concurrent Programming: The Java Programming Language*, 260 pages. Oxford University Press. 1998.

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

[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.

[Kaiser 2003] C. Kaiser and J.F. Pradat-Peyre. Chameneos, a Concurrency Game for Java, Ada and Others. 8 pages, *ACS/IEEE Int. Conf. AICCSA’03*, Tunis, July 2003. IEEE CS Press.

[Lampson 1980] B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23,2. pp. 105-117, February 1980.

[Lea 1997] D. Lea. *Concurrent Programming in Java*. 340 pages. Addison Wesley 1997.

[Lea 1998] D. Lea. Patterns and the democratization of concurrent programming, *IIE Concurrency, Parallel, Distributed & Mobile Computing*, 6,4, pp. 11-13, 1998.

Potratz, Eric. A Practical Comparison Between Java and Ada in Implementing a Real-Time Embedded System. *SIGAda'03 Proceedings*, ACM, pp. 71-83, 2003.

[Quasar 2008]. http://quasar.cnam.fr/

[Redell 1980] D. Redell et al. Pilot : An operating system for a personal computer. *Communications of the ACM*, 23,2. pp. 81-92, February 1980.