Chameneos, a Concurrency Game for Java, Ada and Others

Claude Kaiser, Jean-François Pradat-Peyre
CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris
{kaiser, peyre}@cnam.fr

I. Introduction

This paper presents a peer-to-peer cooperation paradigm and several implementations. The paradigm is expressed as a game. The implementations are done in three different languages largely available to-day (Ada, Java and C with the Posix standard). This allows comparing their programming style and their ability to provide secure programs. Java and Ada are high level languages which allow concurrent programming. Both use the concept of monitor, but their implementation choices make them differ greatly. Posix offers low level system API for concurrent programming.

As prolegomena of the game, we first present the application contexts in which this concurrency paradigm may be useful. Then we summerize the concurrent programming structures of Java, Ada and Posix, and the coloured Petri nets formalism. This presentation may be skipped by the user aware of it.

II. The context

A. Applications requiring concurrent and symmetrical cooperation

More and more applications require that cooperation do not rely only on the client server relationship. They rather specify some form of symmetrical relationship between users. This relationship appears as a symmetrical rendez-vous or peer to peer cooperation. Let us give some examples.

1) In e-commerce, two consumers may decide to join for better condition and to set up internal rules. Their negotiation may start at a meeting infrastructure which can be seen as a virtual marketplace [1].
2) In the internet, peer to peer exchange of data (images, music) starts finding a partner, then agreeing to use a common protocol for data transmission (Napster, Gnutella, Freenet, JXTA).
3) In artificial intelligence multi agent applications, cooperation between agents need some form of negotiation to specialize or modify the agent behaviours [2].

B. Concurrent programming (Java, Ada, others)

Concurrency introduces in the same time design facilities and reliability problems. Indeed, the interleaving of tasks execution leads to a high degree of combinatory and may be the source of subtle mistakes that are difficult to detect by simple simulations or human reasoning. Furthermore, a little modification in a part of the code can produce a major transformation of the application behavior.

In high level languages, such as Java and Ada which allow concurrent programming, concurrency control of asynchronous processes, named “threads” in Java, “tasks” in Ada, relies on the concept of a monitor as introduced by Brinch Hansen [3] and Hoare [4]. A general presentation of concurrent programming is available in [5] and a classification of different monitor implementations is given in [6].

The state variables are encapsulated in the monitor and monitor procedures execute in mutual exclusion. The mutual exclusion is always guaranteed by a lock. The language implementation choices differ by the way of providing condition synchronization and treating queuing rules.

The Java policy uses explicit self-blocking and signaling instructions. It provides “wait()” and “notify()” 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, 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 latters 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.

Ada provides protected object types and 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.

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.

The third concurrency programming tool has been introduced to show how, in the absence of an adequate high level language, a reliable software engineering technique can be implemented by hand, when the underlying operating systems provides the standard Posix interface. Our implementation uses threads that have access to a shared address space (for example a Posix process) and semaphores.

The lock is explicitely implemented with a mutual exclusion semaphore and condition synchronization is implemented with the private semaphore scheme. Recall that the mutual exclusion and private semaphore schemes were introduced by Dijkstra in his seminal paper [Dijkstra 1968] and that the monitor concept is derived from them. The Posix thread and semaphore types use "pthread_t" and "sem_t" structures and "pthread_create()", "pthread_join()", "sem_wait()", "sem_post()", "sem_init()" functions (other Posix types, such as "pthread_mutex" or "pthread_cond_t" could also be used for implementing a monitor like structure; refer to [7]). Another reason for presenting this programming style is that real-time Posix standard are included in Real-Time Java proposals [8].

C. Petri nets and coloured Petri nets

A Petri net [9], [10] is a 4-tuple $(P;T;W^+;W^-)$ where P is the set of places, T is the set of transitions, W^- (resp. W^+) is the the backward (resp. forward) incidence application from $P \times T$ to N.

A Petri net can be viewed as a state transition system where the places denote states or resources of the system and where the transitions denote the actions that model state evolution and resources modification. A marking m of a net is an application from P to N that defines for any place p the number of tokens contained in p for m. The backward incidence application (W^-) reflects for a a place p and a transition t how many instances ($W^-(p,t)$) of token are needed to fire transition t. In the same way, the forward incidence application (W^+) defines how many tokens are produced in place p when firing transition t ($W^+(p,t)$). A transition t is fireable at a marking M if and only if $M(p) \geq W^-(p,t)$ for all places p; the reached marking M' is defined by $\forall p \in P, M'(p) = M(p) - W^-(p,t) + W^+(p,t)$. The set of all reachable markings from the initial marking M_0 is denoted by $Acc(N, M_0)$.

A Petri net is commonly represented by a bipartite valuated graph where nodes are items of $P \cup T$, and arcs are defined by W^+ and W^- in the following way: an arc valued by $n > 0$ exists from a place p to a transition t (resp. from t to p) if and only if $W^-(p,t) = n$ (resp. $W^+(p,t) = n$).

Coloured nets allow the modeling of more complex systems than ordinary nets because of the abbreviation provided by this model. In a coloured net, a place contains typed (or coloured) tokens instead of anonymous tokens in Petri nets, and a transition may be fired in multiple ways (i.e. instantiated). To each place and each transition is attached a type (or a colour) domain. An arc from a transition to a place (resp. from a place to a transition) is labeled by a linear function called a colour function. This function determines the number and the type (or the colour) of tokens that have to be added to or removed from the place upon firing the transition with respect to a colour instantiation.

There are three properties that are fundamental in Petri nets theory : the liveness, the weakly-liveness and the deadlockability. A net is said to be live when, whatever the state reached by the net, all transitions remain fireable in future. A net is said to be deadlockable when it can reach a marking at which no transition is fireable. This marking is called a dead marking and one says that the net has a deadlock. Sometimes, we are sure that the net has no deadlock but we are not sure that the net is live. In this case, we say that the net is weakly-live or deadlock free : at each reachable marking, there is at least one fireable transition.

III. Concurrency paradigm as a game

A. The mutating chameneos

The chameneos existence has been revealed last century in the Solu Khumbu region and figures now in all up-to-date fauna handbooks. Let us quote from one of them [11]:

> Chameneos n [ME.camenious $<$ MFr. cameneon $<$ L. Chameneus $<$ Gr chamaineos $<$ chamai on the ground + neos new] **1.** any of various Old World lizards (family Chamaeoneostidae) which eat honeysuckle leaves, play pall mall, may have blue, yellow or red skin colour, with the property while playing pall mall with a chameneos of a different colour to change its skin colour as well as its partner's one into the third possible colour. **2.** any of various superficially similar reptile that can similarly change colour of their skin, as the Inouït Chameneos (glacialis chamaeneus reptilis) described by J. Malaurie. **3.** a changeable or fickle person-chameneonic *adj.*

The chameneos game is the following: Consider a population of N chameneos that have a cyclic behaviour. A chameneos usually lives lonely eating honeysuckle leaves in the forest and

training. After a while when feeling ready for competition, it enters a mall where a nice spring babbles and where it occasionally plays pall mall with another chameneos and possibly mutates before leaving the mall and returning in the forest. Given an initial population, examine its evolution towards a final state in which all chameneos have the same colour, and therefore in which no one can mutate anymore.

The mutating chameneos is thus a good paradigm for peer-to-peer cooperation of concurrent processes and symmetrical rendez-vous synchronization.

B. *Concurrent behaviour specification*

Each chameneos should respect the following behaviour specification:

1) Asynchronous action before rendez-vous (no interactions).
2) Symmetrical rendez-vous request (at the request the caller does not know whether another chameneos is already present or not, neither if there will be one in some future). This request is sent to a rendez-vous object which acts as a server.
3) Waiting for peer-to-peer rendez-vous.
4) Notification of rendez-vous by the server (a mate is present and its name and colour are known).
5) Cooperative actions which in the chameneos paradigm may lead to colour mutation and to registration of the colour changes. The collaboration may be such that both mates take their part of the work. It may be such that one partner does the job while the other waits for using the results. Any form of cooperation is possible. However the end of this cooperation is a significant event for both partners. The end occurs when each partner has finished and knows that the other has finished, i.e., it is sure that the other does not need any more its cooperation. Both partners may have to wait for it before proceeding.
6) End of synchronous cooperation.

The server, or rendez-vous object, has the following specification:

1) The server must wait until it has received two requests before giving notification.
2) All the requests must be registered and multiple requests shall not disturb the server.
3) Notifications must be sent as soon as possible.
4) When the server notifies A and B, A must known that it mutes with B and B must known that it mutes with A.

The notification pre and post conditions are then:

{the request of A is registered and the request of B is registered}
notification
{A knows B's name and colour and B knows A's name and colour }

Additional warnings are joined to the server specification: the server may be called by more than two chameneos at a time without disturbance.

C. *Concurrent behaviour analysis*

Modeling the server needs to introduce some data that are used to memorize:

- whether the call is the first call of a pair,
- the name and colour of a waiting chameneos,
- the name and colour of the second chameneos.

A possible server behaviour, respecting mutual exclusion, is:

- at first call: register name and colour of the first caller and that the next call will be a second one; wait the end of second call before reading the name and colour of the mate and notifying the rendez-vous to the first caller.
- at second call: register name and colour of the second caller and that the next call will be a first one; read the name and colour of the mate and notify the rendez-vous to the second caller; notice the first caller that its mate name and colour are available.

The corresponding coloured Petri Net is given figure 1. Its

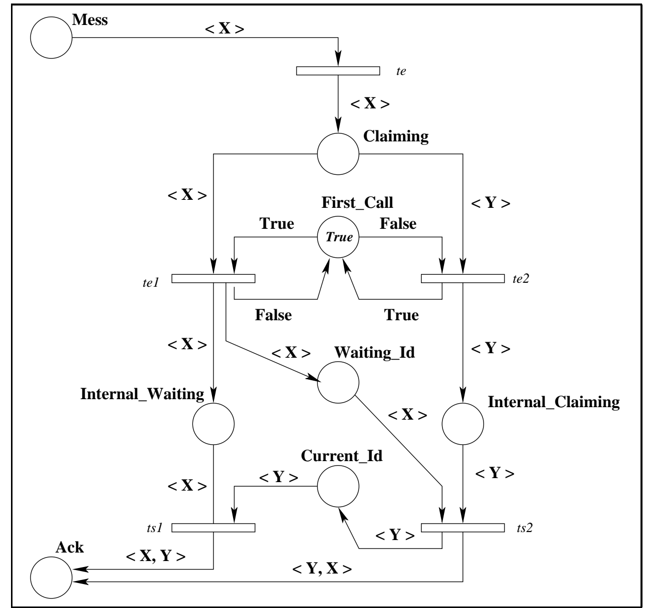

Fig. 1. Coloured Petri net model of the server

analysis shows that there is an inconsistency when the server is late to notify the rendez-vous to the waiting chameneos. For example there is a possible sequence leading to a state where chameneos A expects a rendez-vous with chameneos B, while chameneos B expects a rendez-vous with chameneos C. For instance, A enters the mall, and goes to state `Internal_Waiting` by firing transitions te and $te1$ for $X = A$ (and putting a token A in place `Waiting_Id`); then B comes into the mall and fires te and $te2$; it's then in state (place) `Internal_Claiming`. Suppose that, before A fires $ts1$ and leaves the mall, C comes into the mall and fires te and $te1$ (it puts a token of its value C in places `Internal_Waiting` and `Waiting_Id`). Then B can leave the mall with C (by firing transition $ts2$ for $Y = B$ and $X = C$) while A can leave the mall with B (by firing $ts1$ for $X = A$ and $Y = B$). However there is no deadlock nor

starvation as it can be automatically demonstrated using a tool like Quasar [12], [13].

Previous inconsistency leads to complete the specification and to state that the first call is only possible when the second call is finished (three states are then necessary: a: “first part of first call”, b: “second call”, c: “last part of first call”; and three transitions only are possible: $a \rightarrow b, b \rightarrow c, c \rightarrow a$).

IV. Solutions

A. Using Java objects

In Java, each chameneos is an object which has some behaviour described by a Java thread, and which has some properties such a colour. The mutation is performed individually by each chameneos after it has received a message with the colour of its companion. This mutation can be done outside the mall and is independent of the companion mutation. Thus a chameneos can leave the mall as soon as it receives the message and has not to wait its companion. The corresponding Java objects are given in Annex A.

Synchronized methods are executed in mutual exclusion. However, due to the Java choices of locking and notifying semantic, the entrance of a third chameneos in the mall has to be explicitely forbidden in the program. Otherwise, as demonstrated previously with Petri nets models, an inconsistency may be observed.

B. Using Ada protected objects

The Ada program uses protected objects and requeue statements. Each chameneos manages also its own colour modification. The Ada program is presented in Annex B. The executions of protected object operations are mutually exclusive. Moreover the protected object semantic, called the eggshell model for monitors, gives precedence to already queued entries which have true barriers over other calls contending for the protected object. There is no need to program the interdiction of a third chameneos into the mall.

C. Using semaphores with Posix standards

Using Posix needs a communication between the operating system and the user program. The program text must incorporate library calls performing the correct system calls and must declare data structures for thread and semaphore representations. On the other end, the call parameters must pass the references of these data structures and of the threads codes to allow the system to use them. This communication uses libraries, called Posix binding in Ada (IEEE standard 1003.5) or direct C API. Both are presented in Annex C and D, showing how to use Posix with Ada (using the Florist implementation [14]) and allowing a comparison with the corresponding C code. In principle, a binding with Java may be done similarly. However, we did not find a standard package for doing this.

The cooperation between threads starts by using one semaphore (initialized at the value 2) which aim is to limit to at most two the number of cooperating partners and to block momentarily additional requesting chameneos. This defensive coding forbids any inconsistency due to a third partner. Then the cooperation uses traditionally a mutual exclusion semaphore (initialized at the value 1) and a private semaphore (initialized at the value 0). The latter is used to block the first calling chameneos until the condition of its notification holds. When the second chameneos calls the server, it can be notified immediately since the required data of the first chameneos are already available. Before leaving the server, the chameneos passes the lock to the first chameneos; this uses the synchronization technique called “passing the bâton” [5]. The awaken chameneos, i.e. the first calling chameneos, can now be notified; it is also in charge, before leaving the server, of resetting the server initial state, that is releasing the lock and allowing a new couple of chameneos to start a symmetrical rendez-vous.

V. Comparing the concurrent programming styles when using semaphores, Java or Ada

This case study gives some insights for comparing the concurrent programming style. We add some other aspects deduced from our experience. More general comparison are given in [15]. The three approaches are compared for code simplicity, clarity and reliability and for ease of correctness formal proving.

A. Java strong points (advantages)

The full class and object orientation provides high level language abstractions for the expression of programs and therefore of concurrent objects. The strong typing is a factor of safety.

The existence of a Java virtual machine gives portability, although the language is not standardized. The coexistence of synchronized and not synchronized methods and the possibility of requiring mutual exclusion for a small portion of code only (synchronized block) provide great flexibility.

B. Java weak points (disadvantages)

The choice of having only objects obscures the representation of concurrence and the observation of thread cooperation behaviour. Since Java allows simple inheritance only, a runnable interface has to be implemented by another class (for other Java aspects too, such as graphic interactions or components frameworks, interfaces have to be implemented in order to define environmental supports). When defining subclasses for concurrent objects some impossibility, called inheritance anomaly, can arise [16].

The condition variable with wait, notify, notifyAll is a low level synchronization mechanisms.

Thread scheduling is completely implementation dependant. This imposes to reevaluate the waiting conditions, and to program a waiting loop, thus leading to some form of busy waiting. The barriers (boolean conditions preceeding wait() clauses) are disseminated in the code, allowing to call an already synchronized code in a synchronized code; these mutual monitors calls are susceptible of the well known “nested monitor” deadlock problem .

Last but not least, Java is not standardized and a Java program is not necessarily portable form a version of the language to a new one.

C. Ada strong points (advantages)

Ada provides a high level structural approach for the expression of concurrency, based on explicitly defined active objects. This allows a good visibility of tasks, even in nested scopes, as well as a clear comprehension and observation of the interaction between tasks. The language is standardized, programs are fully portable and the Posix Ada binding is also a standard (IEEE 1003.5).

High level concurrency is provided by barriers, located only at the beginning of an entry code and reevaluated at the end of a subprogram execution. This evaluation and reevaluation is safely done under the protection of the mutual exclusion lock. This leads to simple and readable code. The eggshell model semantic of the protected object is well defined and gives priority to the tasks which are the foremost advanced in the resource usage. With the ceiling-locking scheduling policy, provided with the language real-time annex implementation, mutual calls across protected objects will not deadlock when running on a uniprocessor machine.

D. Ada weak points (disadvantages)

A barrier cannot use the entry call parameters; this complicates the programming of preference control, leading to complicated structures when the requeue statement is used with no assumption about queuing policies. Too many different constructions are available for concurrency: tasks used as servers with a rendez-vous between tasks, protected objects with functions, procedures or entries, low level mechanisms when needed for real-time applications and provided by the language real-time annex.

E. Posix strong points (advantages)

Posix allows a direct action on the underlying operating system; this is supposed to be useful and more efficient for real-time or embedded applications when associated with the control of scheduling policies. This is true with simple hardware architectures (optimization by hand may then be efficient); however this is no longer the case for sophisticated architectures in which caches and pipelines are associated with look-ahead of instruction execution and hardware optimization.

Posix threads (and Posix synchronization mechanisms) are standardized and are implemented in numerous operating systems or real-time executives, and are language independent (in principle).

F. Posix weak points (disadvantages)

Posix provides only low level mechanisms (however Java is not higher level). There is no special linguistic feature for expressing and representing threads; the notion of thread is just a pointer to a supposed sequential code. There is no data encapsulation mechanism, no thread code bracketing (starting and ending a thread execution must be explicitely programmed), no synchronization bracketing (the beginning and the end of mutual exclusion needs to be explicitely marked in the code; leaving a critical section while forgetting a `V` operation, entering a critical section while omitting a `P` operation, are omissions that break the mutual exclusion and makes the code unreliable).

Concurrent programs are difficult to debug since the code and the synchronization are in two different universes with interferences on each others by parameter modifications and passing without type control. The actions required are not clear since many options and parameters are present. Many opportunities for undetected errors are present, the communications between the operating system and the program have many implementation dependencies (for example: exception reporting from operating system to program, exception handling).

VI. Conclusion

A. Game termination

Given an initial population of N Chameneos, the game does not always reaches a final state in which all chameneos have the same colour, and therefore in which no one can mutate anymore. As a matter of fact, according to their initial value and to the population size, the N chameneos population may belong to one of three possible connected components and therefore cannot move from one connected component to another one. Thus the evolution may end only when the initial state is in the same connected components as one final state, else it never stops. For example, if N is a multiple of 3 ($N = 3K$) then the initial state $(1, N-2, 1)$ may lead to termination whilst $(2, N-3, 1)$ doesn't.

B. Real-life considerations

The chameneos paradigm provides a simple example. However it is significant enough to point out the semantic differences and the necessity of defensive code in Java and Posix. Additional comparison of the complete languages may be found in [15]. Providing reliable Posix concurrent programming style may be useful in the future since low-level Posix-like synchronization primitives are included in Real-Time Java proposals [8], aimed for real-time, mobile and embedded applications. Note that high level abstractions like protected objects are also included in these proposals.

We provide also an amusing and interesting paradigm for peer-to-peer communication, which may be extended to peer groups and to real-time considerations.

Acknowledgments

We are grateful to the Ada designers and to the Petri nets community, the efforts of which allow designing better concurrent programs and providing thus more secure real-life computer applications, whatever they are developed in Java, Posix or Ada.

Note

Year 1953, Tensing Norgay and sir Edmund Hillary were the first to succeed in the ascent of mount Everest, also called Chomolungma in Thibetan and Sagarmatha in Nepalese.

REFERENCES

[1] Z. Mamaar, E. Dorion, and C. Daigle, "Toward virtual marketplaces for e-commerce support," *Communications of the ACM*, vol. 44, no. 12, pp. 35–38, 2001.

[2] F. Wolinsky and F. Vichot, "Des multi-agents pour développer des applications de contenu en ligne," *TSI*, pp. 213–232, 2001 (French).

[3] P. Brinch Hansen, *Operating Systems Principles*. Prentice Hall, 1973.

[4] C. A. R. Hoare, "Monitors: an operating system structuring concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[5] G. Andrews, *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[6] P. A. Buhr, M. Fortier, and M. H. Coffin, "Monitor classification," *ACM Computing Surveys*, vol. 27, no. 1, pp. 63–107, 1995. [Online]. Available: citeseer.nj.nec.com/buhr95monitor.html

[7] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages (Third Edition) Ada 95, Real-Time Java and Real-Time POSIX*. Hardback, 2001.

[8] B. Brosgol and B. Dobbing, "Real-time convergence of Ada and Java," in *Proceedings of the 2001 annual ACM SIGAda international conference on Ada*. ACM Press, 2001, pp. 11–26.

[9] W. Reisig, *EATCS-An Introduction to Petri Nets*. Springer-Verlag, 1983.

[10] "Les réseaux de Petri: Modèles fondamentaux," M. Diaz, Ed. Hermès, 2001 (French), no. ISBN : 2-7462-0250-6.

[11] T. N., *Tensing's New World Fauna Handbook, Third Edition*. Tengboche, 1987.

[12] S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau, "Quasar: a new tool for analysing concurrent programs," in *Ada-Europe 2003*, ser. LNCS. Springer-Verlag, 2003.

[13] "Quasar web site," http://quasar.cnam.fr, 2002.

[14] "The fsu implementaton of ieee standard 1003.5b," http://libre.act-europe.fr/GNAT/, 1996.

[15] B. Brosgol, "A comparison of Ada and Java as a foundation teaching language," *ADALTRS: Ada Letters, A Bimonthly Publication of SIGAda, the ACM Special Interest Group on Ada*, vol. 18, 1998.

[16] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1999.

APPENDIX

ANNEX A: JAVA SOLUTION

```
// ========================================================
// class IdChameneos
// ========================================================
public class IdChameneos{
  private final int value;
  public IdChameneos(int val){ value = val; }
  public String toString(){ return value + ""; }
}

// ========================================================
// class Colour
// ========================================================
public class Colour {
  public int internalColour;

  private static final int blueInt   = 0;
  private static final int redInt    = 1;
  private static final int yellowInt = 2;

  public static final Colour BLUE   = new Colour(blueInt);
  public static final Colour RED    = new Colour(redInt);
  public static final Colour YELLOW = new Colour(yellowInt);

  private Colour(int value){ internalColour = value % 3;}

  public Colour ComplementaryColour(Colour C){
    if ( internalColour == C.internalColour )
      return new Colour(internalColour);
    else
      return new Colour( 3 − internalColour − C.internalColour );
  }
  public String toString(){
    if ( internalColour == blueInt )
      return "blue";
    else if ( internalColour == redInt )
      return "red";
    else
      return "yellow";
  }
}

// ========================================================
// class Mall
// ========================================================
public class Mall{
    private Colour AColour, BColour;
    private boolean FirstCall = true;
    private boolean MustWait = false;

    public synchronized Colour Cooperation(IdChameneos x, Colour C){
        Colour result;

        while (MustWait){
            try{wait();} catch(InterruptedException e){}
        }
        if ( FirstCall ){
            AColour = C;
            FirstCall = false;
            while ( ! FirstCall ){
                try{wait();} catch(InterruptedException e){}
            }
            MustWait = false;
            result = BColour;
            notifyAll();
        }
        else{
            BColour = C;
            result = AColour;
            FirstCall = true;
            MustWait = true;
            notifyAll();
        }
        return result;
    }
}

// ========================================================
// class Chameneos
// ========================================================
public class Chameneos extends Thread{
    private Mall mall;
    private IdChameneos id;
    private Colour myColour, otherColour;

    public Chameneos(Mall m, IdChameneos id, Colour c){
        this.mall = m; this.id = id; this.myColour = c;
    }
    private void Message(String Mess){
        System.out.println( "(" + id.toString() + ") I am " +
                            myColour.toString() + " and " + Mess);
    }
    private void EatingHoneysuckleAndTraining(){
        Message("I am Eating Honey suckle and Training");
    }
    private void GoingToTheMall(){
        Message("I am going to the mall");
    }
    private void Mutating(){
        Message("I am going to mute");
        otherColour = mall.Cooperation(id, myColour);
        myColour = myColour.ComplementaryColour(otherColour);
        Message("I have done a mutation");
    }

    public void run(){
        while(true){
            EatingHoneysuckleAndTraining();
            GoingToTheMall();
            Mutating();
        }
    }
}
```

```
// =====================================================
// class Simulation (the main class)
// =====================================================
public class Simulation{

   static Colour[] TheColours =
     { Colour.YELLOW,
       Colour.BLUE,
       Colour.RED,
       Colour.BLUE,
       Colour.YELLOW,
       Colour.BLUE
      };

   static Chameneos[] TheChameneos = new Chameneos[TheColours.length];

   public static void main(String args []) {
      Mall myMall = new Mall();

      for (int i=0; i<TheColours.length; i++){
         TheChameneos[i] =
               new Chameneos( myMall, new IdChameneos(i), TheColours[i] );
      }
      for (int i=0; i<TheColours.length; i++){
         TheChameneos[i]. start ();
      }
   }
}
```

ANNEX B: ADA SOLUTION

```
-- =====================================================
-- package P_Id_Chameneos
-- =====================================================
package P_Id_Chameneos is
   subtype Id_Chameneos is Natural;
end P_Id_Chameneos;

-- =====================================================
-- package P_Colour
-- =====================================================
package P_Colour is
   type Colour is (Blue, Red, Yellow);
   function Complementary_Colour(C1, C2: Colour) return Colour;
end P_Colour;

package body P_Colour is
   function Complementary_Colour(C1, C2: Colour) return Colour is
   begin
      if (C1 = C2) then
         return C1;
      else
         return Colour'Val(3 - Colour'Pos(C1) - Colour'Pos(C2));
      end if;
   end Complementary_Colour;
end P_Colour;

-- =====================================================
-- package Mall
-- =====================================================
with P_Id_Chameneos, P_Colour; use P_Id_Chameneos, P_Colour;
package Mall is
   function Cooperation(X: Id_Chameneos; C: Colour) return Colour;
end Mall;

package body Mall is
   protected Cooperation_Synchro is
      entry Cooperate(X: in Id_Chameneos; C: in Colour; C_Other: out Colour);
   private
      entry Waiting(X: in Id_Chameneos; C: in Colour; C_Other: out Colour);
      First_Call : Boolean := True;
      A_Colour   : Colour;
      B_Colour   : Colour;
   end Cooperation_Synchro;

   protected body Cooperation_Synchro is
      entry Cooperate(X: in Id_Chameneos; C: in Colour; C_Other: out Colour)
      when True is
      begin
         if ( First_Call ) then
            A_Colour := C; First_Call := False;
            requeue Waiting;
         else
            B_Colour := C; C_Other := A_Colour;
            First_Call := True;
         end if;
      end Cooperate;

      entry Waiting(X: in Id_Chameneos; C: in Colour; C_Other: out Colour)
      when First_Call is
      begin
         C_Other := B_Colour;
      end;
   end Cooperation_Synchro;

   function Cooperation(X: Id_Chameneos; C: Colour) return Colour is
      Other_Colour : Colour;
   begin
      Cooperation_Synchro.Cooperate(X, C, Other_Colour);
      return Other_Colour;
   end Cooperation;
end Mall;

-- =====================================================
-- package P_Chameneos
-- =====================================================
with P_Id_Chameneos, P_Colour; use P_Id_Chameneos, P_Colour;
package P_Chameneos is
   task type Chameneos is
      entry Start (Id: in Id_Chameneos; C: in Colour);
   end Chameneos;
end P_Chameneos;

with Text_IO, Mall; use Text_IO, Mall;
package body P_Chameneos is

   task body Chameneos is
      My_Id                   : Id_Chameneos;
      My_Colour, Other_Colour : Colour;

      procedure Message(Mess : in String ) is
      begin
         Put_Line("(" & Id_Chameneos'Image(My_Id) & ") I am " &
                  Colour'Image(My_Colour) & " and " & Mess);
      end;
      procedure Eating_Honey_Suckle_And_Training is
      begin
         Message("I am eating honey suckle and training");
      end;
      procedure Going_To_The_Mall is
      begin
         Message("I am going to the mall");
      end;
      procedure Mutating is
      begin
         Message("I am ready to mute");
         Other_Colour := Cooperation(My_Id, My_Colour);
         My_Colour := Complementary_Colour( My_Colour, Other_Colour);
         Message("I have performed a mutation");
      end;
   begin
      accept Start (Id: in Id_Chameneos; C: in Colour) do
         My_Id := Id; My_Colour := C;
      end Start;
      loop
         Eating_Honey_Suckle_And_Training;
         Going_To_The_Mall;
         Mutating;
      end loop;
   end Chameneos;
end P_Chameneos;

-- =====================================================
-- procedure Simulation (main procedure)
-- =====================================================
with P_Id_Chameneos, P_Colour, Mall, P_Chameneos;
use  P_Id_Chameneos, P_Colour, Mall, P_Chameneos;
procedure Simulation is
   The_Colours : array(Natural range<>) of Colour :=
     (Yellow, Blue, Red, Blue, Yellow, Blue);
   The_Chaemenos : array(The_Colours'Range) of Chameneos;
begin
   for I in The_Chaemenos'Range loop
      The_Chaemenos(I).Start ( Id_Chameneos(I), The_Colours(I) );
   end loop;
end Simulation;
```

ANNEX C: POSIX STYLE SOLUTION WITH THE FLORIST POSIX/ADA BINDING

```
with Posix, Posix.Semaphores; use Posix, Posix.Semaphores;

package body Mall_Posix is
   AtMostTwo : Semaphore;
   Mutex     : Semaphore;  -- mutual exclusion semaphore or lock
   SemPriv   : Semaphore;  -- private semaphore used to pass the baton

   FirstCall  : Boolean := True;
   AColour    : Colour;
   BColour    : Colour;

   procedure P(S: in Semaphore) is begin Post( Descriptor_Of(S) ); end P;

   procedure V(S: in Semaphore) is begin Wait( Descriptor_Of(S) ); end V;

   function Cooperation(X: Id_Chameneos; C: Colour) return Colour is
      Other_Colour : Colour;
   begin
      P(AtMostTwo); -- limits the number of partners
      P(Mutex);
      -- user programmed mutual exclusion = setting the lock
      if FirstCall then
         AColour := C; FirstCall := False;
         -- the next call will be considered as a second one
         V(Mutex); P(SemPriv); -- waiting for the lock
         Other_Colour := BColour;
         V(Mutex); -- releases the lock since the rendez-vous ends
         V(AtMostTwo); V(AtMostTwo); -- allows a new pair
      else -- this is the second chameneos of the pair
         FirstCall := True;
         BColour := C;
         Other_Colour := AColour;
         -- the next call will start a new meeting
         V(SemPriv); -- passes the lock to its mate
      end if;
      return Other_Colour;
   end Cooperation;

begin
    Initialize (AtMostTwo, 2);
    Initialize (Mutex, 1);
    Initialize (SemPriv, 0);
end Mall_Posix;
```

ANNEX D: PURE POSIX SOLUTION

```
/* ========================================================== */
/* file  types.h                                               */
/* ========================================================== */
#define NB_CHAMENEOS 4
typedef int idChameneos;
typedef enum {Blue, Red, Yellow} colour;

/* ========================================================== */
/* file  cooperation.c                                         */
/* ========================================================== */
#include <semaphore.h>
#include "types.h"

sem_t AtMostTwo;
sem_t Mutex;
sem_t SemPriv;

int     FirstCall  = 1;
colour  AColour;
colour  BColour;

/* ========================================================== */
colour Cooperation(idChameneos id, colour c){
  colour otherColour;
  int val;

  sem_wait(&AtMostTwo); // limits the number of partners
  sem_wait(&Mutex);
  // user programmed mutual exclusion = setting the lock
  if ( FirstCall ) {
    AColour = c; FirstCall = 0;
    // the next call will be considered as a second one
    sem_post(&Mutex); sem_wait(&SemPriv); // waiting for the lock
    otherColour = BColour;
    sem_post(&Mutex); // releases the lock since the rendez-vous ends
    sem_post(&AtMostTwo); sem_post(&AtMostTwo); // allows a new pair
  }
  else { // this is the second chameneos of the pair
    FirstCall = 1;
    BColour = c;
    otherColour = AColour;
    // the next call will start a new meeting
    sem_post(&SemPriv); // passes the lock to its mate
  }
  return otherColour;
}

/* ========================================================== */
void initCooperation (void){
  sem_init(&AtMostTwo, 0, 2);
  sem_init(&Mutex, 0, 1);
  sem_init(&SemPriv, 0, 0);
}

/* ========================================================== */
/* file  simulation.c                                          */
/* ========================================================== */
#include <stdio.h>
#include <pthread.h>
#include "types.h"

/* ========================================================== */
colour complementaryColour(colour c1, colour c2){
  if ( c1 == c2)
    return c1;
  else
    return (3-c1-c2);
}

/* ========================================================== */
extern colour Cooperation(idChameneos id, colour c);
extern void   initCooperation (void);

/* ========================================================== */
void chameneosCode(void *args){
  idChameneos myId;
  colour      myColour, oldColour, otherColour;

  sscanf((char *) args, "%d %d", &myId, &myColour);

  printf("(%d) I am (%d) and I am running\n", myId, myColour);

  while (1){
    printf("(%d) I am (%d) and I am eating honey suckle and training\n",
           myId, myColour);
    printf("(%d) I am (%d) and I am going to the mall \n",
           myId, myColour);
    otherColour = Cooperation( myId, myColour);
    oldColour = myColour;
    myColour = complementaryColour( myColour, otherColour);
    printf("(%d) I am (%d) and I was %d before\n",
           myId, myColour, oldColour);
  }
}

/* ========================================================== */
int main(void){
  colour tabColour[NB_CHAMENEOS] = {Yellow, Blue, Red, Blue};
  char theArgs[255][NB_CHAMENEOS];
  pthread_t  tabPid[NB_CHAMENEOS];
  int i;

  initCooperation ();

  for (i=0; i<NB_CHAMENEOS; i++){
    sprintf( theArgs[i], "%d %d", i, tabColour[i]);

    pthread_create(&tabPid[i], NULL, (void *(*)()) chameneosCode, theArgs[i]);
  }
  // waiting the end of children (that will never come)
  for (i=0; i<NB_CHAMENEOS; i++){
    pthread_join(tabPid[i], NULL);
  }
  return 0;
}
```