Comparing the Reliability Provided by Tasks or Protected Objects for Implementing a Resource Allocation Service : a Case Study.

C.Kaiser and J.F. Pradat-Peyre
Conservatoire National des Arts et Métiers
Laboratoire CEDRIC
kaiser@cnam.fr, peyre@cnam.fr

Abstract

We compare two possible implementations of a resource allocation service, one using a task server, the other using a protected object. Both make use of the requeue statement, the count attribute, and also the abort statement in order to satisfy requests, depending on the parameters passed in by the calling task and on the internal state of the service. Because the schema of requeue and entries has an execution semantic based on state and transition, it can be coupled easily with a proof in terms of colored Petri nets. We consider the dining philosophers problem, which is a good illustration of the need for a resource allocation service and for which deadlock- and starvation-free implementations have already been given in Ada95, though not formally proven and sometimes unfair. We give an almost forgotten solution where the dining philosophers problem is safely implemented with protected objects, whereas its implementation with a server task leads to deadlock. We provide two implementations, one of which completes a solution presented by Brosgol in Ada Europe 96 and makes it really fair. Informal proofs are given and are confirmed by Petri nets proofs. Through these examples, we show that the eggshell semantics of protected objects are basic for attaining a reliable implementation.

1 Introduction

1.1 Resource Allocation

A resource allocation policy must take care of clients' requirements for quality of service (efficiency of allocation management, availability of resources during a given time period); it must also guarantee global invariant properties of the system behavior such as the absence of deadlocked clients, of infinitely postponed clients, and of clients never releasing their allocated resources (or simply not releasing the resources in the contractual delay).

We consider resource allocation controlled by a service implemented in Ada95. We treat deadlock and starvation prevention as well as client respect of deadlines. Deadlock may occur if the opportunity is left to successive task requests to empty the pool of resources without satisfying one task completely; thus, all tasks still require more resources while refusing to release a part of their loan. The reliability approach must be global because preventing deadlock may introduce starvation and preventing starvation may reintroduce deadlock.

1.2 Presentation of our Approach

Two possible implementations of a resource allocation service, one using a task server, the other using a protected object are compared. Both make use of the requeue statement, the count attribute and also the abort statement.

The entry selection together with the requeue statement provides the ability to satisfy a request depending on the parameters passed in by the calling task and also on the internal state of the server. This is sometimes called preference control [Int95].

According to our experience, the underlying automaton-like structure chosen for the semantics of the requeue statement favors a design process mixing Ada95 program construction and Petri net validation of this construction. The presented work has been done by alternating between the design and the validation.

We show that the eggshell semantics of protected objects lead to a more reliable implementation than the task server solution. For this, we give an example where a resource allocation algorithm is safely implemented with protected objects, whereas its implementation with a server task leads to deadlock (this deadlock, which is not likely to be observed, is easily detected by formal methods).

2 The Dining Philosophers Problem

2.1 Statement of the Problem and Former Ada Implementations

To proceed, a task often needs the presence of a couple of resources of different classes: for instance a channel with each of two adjacent tasks, a printer and a disk drive to print a stored file,...

The dining philosophers is a good illustration of such a need. As stated by B.Brosgol in [Bro96] *"The Dining Philosophers example is a classical exercise for concurrent programming. Originally posed by Dijkstra [Dij71], the problem may be stated as follows, generalized to allow an arbitrary number of philosophers: For an arbitrary integer N greater than 1, there are N philosophers seated around a circular table. In front of each philosopher is a plate of food, and between each pair of philosophers is a chopstick. The "processing" performed by each philosopher is an endless iteration of the two actions Eat and Think. In order to perform the Eat action, a philosopher needs two chopsticks: in particular the one immediately to the left and the one immediately to the right. (Thus at most only N/2 philosophers can eat simultaneously). Design a solution so that for an arbitrary integer M, each philosopher is guaranteed to perform Eat-Think sequence (at least) M times."*

Different Ada95 implementations of this problem can be found. [BW95] gives a deadlock free and fair (only for FIFO entry queues) solution with $N + 1$ server tasks. [BKPP97] implements the same solution with a single protected object that hides the service states from the clients and the authors prove the correctness and fairness of this solution (for any queuing policy). [Bro96] provides two solutions, both with a protected object; their implementations are deadlock free, but none prevents starvation.

2.2 The Solution Implemented in this Paper

The eating condition defined by Dijkstra in the first published solution [Dij71] was not stated in terms of the availability of chopsticks. Rather, the problem statement says that the philosophers can be thinking, hungry, or eating, and that a philosopher X can eat if and only if none of its neighbors, $X - 1$ and $X + 1$, is eating (they can be thinking or hungry; in both cases, they have no chopstick allocated). The eating condition is:

$$(X - 1 \text{ is not eating}) \text{ and } (X + 1 \text{ is not eating }).$$

This approach does not prevent starvation. Courtois and Georges showed [CG77] that a solution preventing starvation as well as deadlock can be constructed on the basis that a philosopher X can eat if the left neighbor (i.e. $X - 1$) does not eat and also is not hungry. The condition is now:

$$(X - 1 \text{ is thinking}) \text{ and } (X + 1 \text{ is not eating }).$$

In the next sections we compare the implementations of this solution using tasks or protected objects. We give dual implementations : one with philosopher status as above; the other with chopsticks status in order to complete the second solution presented by Brosgol in Ada Europe 96 and to make the solution really fair.

2.3 Working Properly in the Presence of Requeue with Abort

We add a deadline to the Eat action of a philosopher. When the philosopher deadline is reached, its chopsticks request or its chopsticks allocation is aborted, even if a request is postponed after being requeued. This is possible in Ada95 by using the "requeue with abort" clause. All chopsticks are returned to the pool, at the latest when the Eat action is aborted.

3 Ada95 Programming

The complete Ada95 programs corresponding to the solutions given in this paper can be found at the ftp site : "ftp://bacchus.cnam.fr/pub/articles/TriAda97/programs".

There are two versions: one focuses on the comparison between the use of protected objects or tasks for the implementation of the resources allocation server, the other provides a complete simulation environment allowing us to analyze performance of solutions. Both are composed of five packages.

An APPLICATION package groups the set of PHILOSOPHERS tasks that have a common cyclic behavior. During a cycle, a philosopher thinks, then requests, uses and releases chopsticks. The time allowed for requesting and using chopsticks is limited by a deadline which expiration is signaled to the protected object SUPERVISION.CONTROL which triggers the asynchronous transfer of control. The philosopher then returns its chopsticks. The allocation, usage and release of chopsticks are encapsulated in a procedure named transaction which assigns the deadline. The SUPERVISION package groups the protected object CONTROL and a task for time service. The ALLOCATION package contains the task or the protected object SERVER which is the shared service used by all philosophers. The SCENE package provides a simulated environment for philosophers and the package COMMON contains the constants common to all packages.

3.1 Source Code of the Simpler Version

3.1.1 The Package COMMON

```
-------------------------------------------------------
package common is
   N : constant integer := 5;  -- number of philosophers
   type philo_id is mod N;
end common;
-------------------------------------------------------
```

3.1.2 The Package SCENE

As the implementation of this package does not present any interest we give here only its specification (the body is provided in Appendix).

```
-------------------------------------------------------
with common; use common;
package scene is
   procedure thinking(x : in philo_id);
   procedure eating(x : in philo_id);
   function random_duration (d1,d2 : in duration)
                                    return duration;
end scene;
-------------------------------------------------------
```

3.1.3 The Package SUPERVISION

This package provides the protected object `control` used to supervise a transaction.

When `control.start_step` is called, a timer associated to x is set and its expiration opens the protected entry `control.stop_me` which was initially closed. The procedure `control.end_of_step` cancels the previous timer and closes the protected entry `stop_me`.

The complete specification and body of this package can be found at the ftp site mentioned above.

```
-------------------------------------------------------
package Supervision is
   protected Control is
      procedure start_step(x : in philo_id; date: time);
      procedure end_of_step(x : in philo_id);
      entry stop_me(philo_id);
      ...
   end Control;
   ...
end Supervision;
-------------------------------------------------------
```

3.1.4 The Procedure TRANSACTION

A transaction consists in requesting resources (here the two chopsticks), eating and, releasing the resources. The requesting and the eating actions are processed with a deadline d.

```
-------------------------------------------------------
with Ada.Calendar,Common,Allocation,Supervision,Scene;
use Ada.Calendar,Common,Allocation,Supervision,Scene;

procedure Transaction_With_Abort(me : in philo_id;
                                 d : in duration) is
   -- d is the critical delay
   deadline : time;
begin
   deadline := Ada.Calendar.clock + Duration(d);
   -- giving a deadline and requiring supervision
   control.start_step(me, deadline);
   select
      control.stop_me(me);
      -- the deadline is over; temporal fault
   then abort
      -- this is the abortable part of the transaction
      request(me); -- requests the chopsticks
      eating(me);  -- eating action
   end select;
   -- stops the deadline supervision
   control.end_of_step(me);
   -- releases chopsticks after action or after deadline
   release(me);  -- release the chopsticks
end Transaction_With_Abort;

procedure Transaction(me : in philo_id;
                      d : in duration) is
begin
   request(me); -- requests the chopsticks
   eating(me);  -- eating action
   release(me); -- release the chopsticks
end Transaction;
-------------------------------------------------------
```

3.1.5 The Procedure APPLICATION

The procedure Application is the main procedure of the program. The philosophers are tasks, and in order to name the philosophers, we use a discriminant (`X:Philo_Id`) in the declaration of the task type. Its default inititialization results of the call to function `Unique_Id`.

Each philosopher loops forever being alternatively thinking (call to `Scene.Thinking`) or performing a transaction.

```
-------------------------------------------------------
with Common, Transaction, Scene;
use Common;

procedure application is

   Next_Id : Philo_Id := Philo_Id'last;
   function Unique_Id return Philo_Id is
   begin
      Next_Id := Next_Id + 1;
      return Next_Id;
   end Unique_Id;

   task type philo( X: Philo_id := Unique_Id);
   philosopher : array(philo_id) of philo;

   task body philo is
      d : duration; -- critical delay before deadline
   begin
      loop
         Scene.Thinking(X);
         d := Scene.Random_Duration(0.5, 0.8);
         Transaction_With_Abort(X, d);
      end loop;
   end Philo;

begin
   null;
end Application;
-------------------------------------------------------
```

3.1.6 The Package ALLOCATION Using a Protected Object

```
--------------------------------------------------------
with common; use common;
package allocation is
   procedure request(me : in philo_id);
   procedure release(me : in philo_id);
private
   type THEA is (thinking, eating);
   type THEA_array is array(philo_id) of THEA;
end allocation;  -- end of package declarations

package body allocation is

   protected server is
      entry request(philo_id);
      procedure release(x : in philo_id);
   private
      status : THEA_array := (others => thinking);
   end server;

   procedure request(me : in philo_id) is
   begin
      server.request(me);
   end request;
   procedure release(me : in philo_id) is
   begin
      server.release(me );
   end release;

   protected body server is
      entry request(for x in philo_id)
      when status(x + 1) = thinking and
        status(x - 1) = thinking and
        request(x - 1)'count = 0 is
      begin
         status(x) := eating;
      end request;
      procedure release(x : in philo_id) is
      begin
         -- x is eating when not aborted
         -- x may be thinking if aborted before
         -- calling request
         -- x may be hungry if aborted when
         -- queuing for request
         status(x) := thinking;
      end release;
   end server;

end allocation;  --end of the package body
--------------------------------------------------------
```

3.1.7 The Package ALLOCATION Using a Task

```
--------------------------------------------------------
with Common; use Common;
package Unreliable_Allocation_Philo is
   procedure request(me : in philo_id);
   procedure release(me : in philo_id);
private
   type THEA is (thinking, eating);
   type THEA_array is array(philo_id) of THEA;
end Unreliable_Allocation_Philo;

package body Unreliable_Allocation_Philo is

   task Server is
      entry Request(Philo_Id);
      entry Release(X : in Philo_Id);
   end Server;

   procedure Request(Me : in Philo_Id) is
   begin
      Server.Request(Me);
   end Request;

   procedure Release(Me : in Philo_Id) is
   begin
      Server.Release(Me);
   end Release;

   task body Server is
      Status : THEA_Array := (others => Thinking);
   begin
      loop
         select
            when Status(1)=Thinking and
              Status(4)=Thinking and
              Request(4)'Count=0 =>
               accept Request(0) do
                  Status(0) := Eating;
               end Request(0);
         or
            ....
         or
            when Status(0)=Thinking and
              Status(3)=Thinking and
              Request(3)'Count=0 =>
               accept Request(4) do
                  Status(4) := Eating;
               end Request(4);
         or
            accept Release(X : in Philo_Id) do
               -- x is eating when not aborted
               -- x may be thinking if aborted before
               -- calling request
               -- x may be hungry if aborted when
               -- queuing for request
               Status(X) := Thinking;
            end Release;
         or
            terminate;
         end select;
         delay(Duration(0.5));
           -- in order to make the deadlock more
           -- likely to occur
      end loop;
   end Server;

end Unreliable_Allocation_Philo;

--------------------------------------------------------
```

4 The Validation by Colored Petri-Nets

Ada83 tasks and the rendez-vous statement have already been modeled by means of Petri nets in the past [MSS89, TSM90]. In this paper, we extend the modeling to protected object, requeue and abort statement and we use extensively new results and available tools that allow us to model more realistic situations. Among these results one can quote colored reductions [Had91, BHPP97] and structure theory based on the controlled siphon property [BPP96]. We are using tools [1] such as GreatSPN, CPN-AMI or DesignCPN.

Our validation process follows three steps:

1. the complete program is modeled using a hierarchical approach (for instance using DesignCPN);
2. the model obtained is then reduced by applying colored Petri nets reductions (with for instance CPN-AMI) while preserving fundamental properties of the model (such as deadlock freeness, fairness, ...);
3. finally, the reduced model is analyzed for proving (or disproving) the absence of deadlock and starvation using principally a characterization of liveness by means of the controlled siphon property and structural invariants.

This methodology reduces the complexity of the analysis and allows us to treat realistic situations. In this paper, the whole program structure is analyzed including the asynchronous transfer of control.

In particular, we prove that the solution using a protected object to implement the resource allocator is fair and deadlock free while the solution using a task is not deadlock free.

4.1 Petri Nets and Colored Petri Nets

A Petri net [Rei83] is a 4-tuple $\left\langle\ P, T, W^+, W^-\ \right\rangle$ 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 $\mathbb{N}$.

A Petri net can be viewed as a state transition system where the places denote some kind of tokens and the transitions the actions that produce and/or consume tokens. A marking of a net is an application from P to $\mathbb{N}$ that defines for any place p the number of tokens of kind p. The backward incidence application (W^-) reflects for a kind of token (a place p) and an action (a transition t) how many instances ($W^-(p,t)$) of this token are needed to do this action (to fire the transition t). In the same way, the forward incidence application (W^+) defines how many instances of a kind of token p are produced by an action 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 place p. The marking M' reached by the firing of t at marking M is defined by $\forall p \in P, M'(p) = M(p) - W^-(p,t) + W^+(p,t)$. The set of all accessible 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. $W^+(p,t) = n$). One notes ${}^\bullet p$ (resp. $p^\bullet$) the set of transitions such that there exists an arc from these transitions to p (resp. from p to these transitions): ${}^\bullet p = \left\{t \in T | W^+(p,t) > 0\right\}$ and $p^\bullet = \left\{t \in T | W^-(p,t) > 0\right\}$.

Three properties are fundamental in Petri nets theory: the liveness, the deadlock-freeness and the deadlock-ability.

A net is said to be **live** when, whatever the state reached by the net, all transitions remain fireable in future: $\forall m \in Acc(N, M_0), \forall t \in T, \exists m' \in Acc(N, m) \,|\, m'[t >$.

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: $\exists m \in Acc(N, M_0) \,|\, \forall t \in T, m[t \not>$.

A non deadlockable net is said to be **deadlock free**. At each reachable marking, we insure that at least one transition is fireable: $\forall m \in Acc(N, M_0), \exists t \in T \,|\, m[t >$.

Colored nets allow the modeling of more complex systems than ordinary ones because of the abbreviation provided by this model. In a colored net, a place contains typed (or colored) 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 color) domain. An arc from a transition to a place (resp. from a place to a transition) is labeled by a linear function called a color function. This function determines the number and the type (or the color) of tokens that have to be added or removed to or from the place upon firing the transition with respect to a color instantiation. These different concepts can be formalized by the following definitions.

Definition 1 *A colored Petri net (or colored net) is a 6-tuple $CN =< P, T, \mathcal{C}, W^-, W^+, M_0 >$ where:*

- *P is the set of places, T is the set of transitions with $(P \cap T = \emptyset, P \cup T \neq \emptyset)$*
- *$\mathcal{C}$ is the color function from $P \cup T$ to Ω, where Ω is a set of finite non empty sets. An item of $\mathcal{C}(s)$ is called a color of s and $\mathcal{C}(s)$ is called the color domain of s.*
- *W^+ (reps. W^-) is the forward (resp. backward) incidence matrix defined on $P \times T$ where $W^+(p,t)$ and $W^-(p,t)$ are linear applications from $Bag(C(t))$ to $Bag((C(p))$.* [2] *The incidence matrix of the net is*

[1] for more informations on these tools, please refer to the url http://www.daimi.aau.dk/PetriNets/tools/ quick.html

[2] if A is a finite and non empty set, then $Bag(A)$ denotes the set of multi-sets (i.e. sets that may include multiple occurrences of the same item)over A

defined by $W = W^+ - W^-$.

- *M_0 is the initial marking of the net and is an application defined on P whith $M_0(p) \in Bag(C(p))$.*

Definition 2 *Let $CN = \langle P, T, W^+, W^-, M_0 \rangle$ be a colored net. A marking of CN is a vector indexed by P with $\forall p \in P, M(p) \in Bag(C(p))$. A transition t is fireable for a color $c_t \in C(t)$ and for a marking M if and only if: $\forall p \in P, M(p) \geq W^-(p,t)(c_t)$. The reached marking M' is defined by $\forall p \in P, M'(p) = M(p) - W^-(p,t)(c_t) + W^+(p,t)(c_t)$. One note, $M[t >_{c_t} M'$.*

Generally, color domains are compositions of basic ones, called *classes*, and color functions are tuple of basic functions defined on these classes. A class is a finite and non empty set and its size may be parameterized by an integer. The particular class ϵ contains the only item $\bullet$: $\epsilon = \{\bullet\}$.

The color functions used on our models are the identity (or selection) denoted by X or d, the successor mapping denoted by $X{+}{+}$ and the predecessor mapping denoted by $X{-}{-}$. If D is a domain, $D.All$ denotes all the token of D.

4.2 Modeling by Means of Colored Petri Nets

The previous Ada program can be modeled using colored Petri nets. We give the model in three parts: we first model the global structure of the program (Fig. 1); second, we discuss how the resources are allocated by the server (Fig. 2); and third, we explain how the resources are released (Fig. 3).

The complete model is obtained by merging these three subnets and is depicted in Appendix (Fig. 5).

In these models, dashed transitions are priority transitions, meaning that when there is a conflict between two transitions, preference is given to the transition with the highest priority. We use these semantics to take into account the Ada95 processing policy of protected objects and of the abort instruction: at the end of a protected call, already queued entries (whose barriers are true) take precedence over new calls, and an abort sequence takes precedence over new call.

We use three levels of priority : a level 0 for nonpriority transitions, denoted in the graph by white transitions, a level 1 for transitions regarding the internal actions of the protected object server, denoted in the graph by gray transitions, and a level 2 for transitions regarding the abort sequence, denoted by black transitions.

4.2.1 The Model of the Global Program Structure

This net is composed of two parts: one deals with the evolution of the philosophers, the other (drawn with dashed line) concerns the abort part of a transaction.

The philosophers can be in one of the following states, modeled in the net by the gray places :

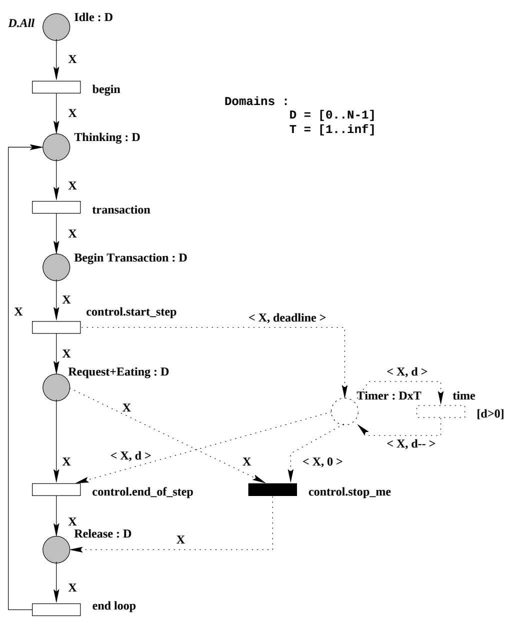

Figure 1: The model of the global program structure with the asynchronous transfer of control

- *Idle* : the initial state of all philosophers (modeled in the net by $M_0(Idle) = D.All$).
- *Thinking* : a philosopher who is calling `thinking`.
- *BeginTransaction* : ready to perform a transaction
- *Request + Eating* : performing a call to `request` and then to `eating`
- *Release* : performing a call to `release`.

A philosopher x becomes *Thinking* by firing transition *begin.* It can then fire transition *transaction* and enter state *BeginTransaction.* From this state it can fire the transition *control.start_step* and enter state *Request + Eating.* It enters then state *Release* by firing either the transition *control.end_of_step* or the transition *control.stop_me.* At the end it returns to state *Thinking.*

A part of a transaction can be aborted if the associated deadline expires. Place *Timer* models the time remaining for a philosopher x to perform its transaction. When transition *control.start_step* is fired (by a philosopher x), a token $(x, deadline)$ is added to place *Timer.* Transition *time* becomes fireable and can then decrease the timer associated with x (each firing of transition *time* decrease the timer by 1 : a token (x, d) is replaced by a token $(x, d-1)$ in place *Timer*). This transition remains fireable until either the deadline has expired ($d = 0$) and the philosopher x has fired transition *control.stop*, or the transaction has ended and the philosopher x has fired transition *control.end_of_step*.

4.2.2 The Model of the Request Action

We take a look now on the subnet (Fig. 2) modeling the allocation policy used by the protected object server. This model details the state *Request* + *Eating* of the previous model.

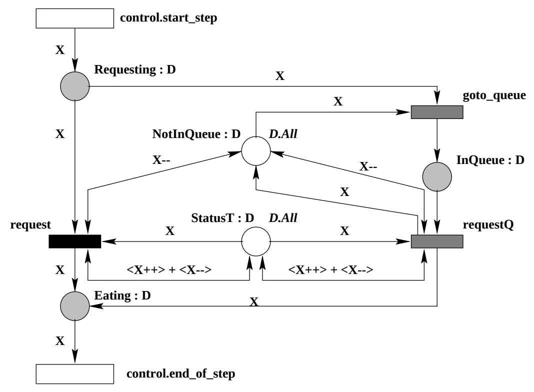

Figure 2: The model of the request action

When performing a call to `server.request` or performing a call to `scene.eating`, a philosopher x can be in one of the three following states:

- *Requesting* : x is evaluating the guards associated to the entry `request`
- *InQueue* : x has been queued on the entry `request`
- *Eating* : x has been served and is calling `eating`

The guard associated with the entry `server.request` examines whether or not some philosophers are *Thinking*. Thus, the place *StatusT* models the status variable with the following meaning: $M(StatusT(x)) = 1$ if and only if the philosopher x is thinking. Initially, all philosophers are thinking ($M_0(StatusT) = D.All$).

Place *NotInQueue* models the absence of philosophers in the queue of the entry *request*. Initially, no philosophers are in this queue and $M_0(NotInQueue) = D.All$.

This subnet can be interpreted as follow: after firing transition *control.start_step*, a philosopher x enters state *Requesting*. From this state, it can fire transition *request* if $x - 1$ (more precisely the result of the mapping $X--$ on x) is not in the queue (the test arc between *request* and *NotInQueue* with valuation $X--$), and if $x + 1$ and $x - 1$ are thinking (because of the test arc between *request* and *StatusT* valuated by the mapping $< X++ > + < X-- >$). In this case, it enters state *Eating* and a token x is removed from place *StatusT*, meaning that x is now eating. If *request* is not fireable, then x fires *goto_queue*, enters state *InQueue* and a token x is removed from place *NotInQueue*, meaning that x is now in the queue.

As for the transition *request*, a philosopher x in state *InQueue* can fire transition *requestQ* if $x - 1$ is not in the queue and $x + 1$ and $x - 1$ are thinking. In this case it enters the state *Eating* and a token x is added to place *NotInQueue*, meaning that x is no longer in the queue.

Note that in this subnet, the transition *request* has priority over the transition *goto_queue*, meaning that x enters *InQueue* if and only if it cannot fire *request*.

The abort part is now decomposed in two parts depending on the state of the philosophers (*Eating* or *InQueue*) and can be viewed in the global model (Fig. 5).

4.2.3 The Model of the Release Action

The last subnet concerns the release action and is shown in Figure 3. It details the state *Release* of the first model.

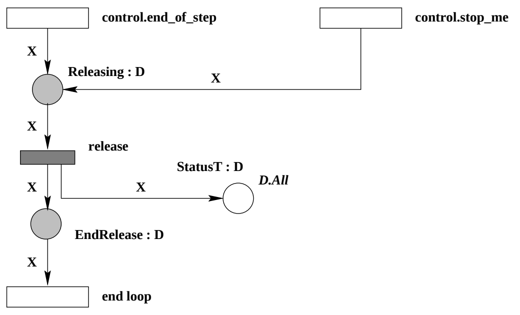

Figure 3: The model of the release action

When performing a call to `server.release`, a philosopher x can either be in state *Releasing* (x is ready to call the procedure `release`) or be in state *EndRelease* (it has just called `release`).

In state *Releasing*, a philosopher x can fire transition *release* and enters state *EndRelease*. In this case, a token x is added to place *StatusT* (the same place than in the previous model), meaning that x is now thinking again.

4.3 Proving Correctness of the Complete Model

The complete model (Fig. 5) can be automatically reduced using reduction theory of Petri nets[Ber86, Had91] while preserving properties of the initial model (checking the liveness or the absence of starvation in the reduced net is equivalent to checking these properties in the original one). We obtain the net depicted in Figure 4.

One can remark that some transitions have been merged: the transition *transaction* with *control.start_step*, and the

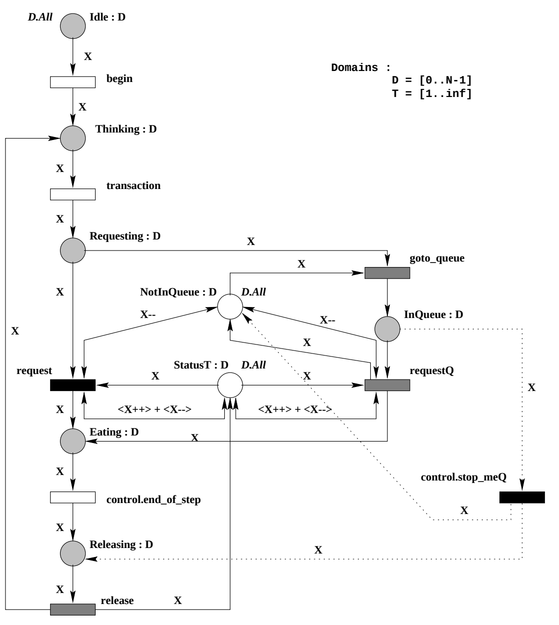

Figure 4: The reduced net of the dining philosophers

transition *release* with *endloop*. Intermediate states have been suppressed (*BeginTransaction* and *EndRelease*).

A less trivial reduction associates each instance of transition *time* with its successor ($time(x, d)$ with $time(x, d{+}{+})$), making possible a aggregation of the transition $time(0)$ with the transitions *control.end_step*, *control.stop_meA* and *control.stop_meQ*. After this reduction, place *Timer* becomes redundant and is suppressed while the two transitions *control.end_step* and *control.stop_meA* become identical and are merged.

This last reduction can be interpreted as a compression of the time: the time is now infinitely fast : the deadline expires at the moment the timer is set.

4.3.1 Proving Deadlock Freeness of the Model

A siphon S is a subset of places such that each transition that puts a token in S also needs and consumes some tokens in S (${}^\bullet S \subseteq S^\bullet$). Thus, as soon as a siphon is empty of tokens, all transitions of $S^\bullet$ are dead and therefore the net is not deadlock free. A siphon is said to be controlled if it can never become empty of tokens. The theory [BPP96] proves that if all the siphons are controlled, then the net is deadlock free. Therefore we have to prove that all siphons of the net are controlled.

Without the Abort Part

In the net of Figure 4, there are several siphons but only one is not controlled: the siphon $D = \{NotInQueue\}$. Indeed, suppose that the philosopher 0 is eating and that the philosopher 1 fires *transaction*. By construction, this philosopher goes to state *InQueue* (because 0 is eating) and becomes hungry (the token $x = 1$ has been removed from place *NotInQueue*). Let us suppose now that the philosopher 2 calls *transaction*. Because 1 is hungry, 2 also goes to state *InQueue* and also becomes hungry. If all other philosophers (from 3 to $N - 1$) also fire *transaction*, they will all become hungry (only the token $x = 0$ remains in place *NotInQueue*).

Let us suppose now that philosopher 0 fires successively transitions *control.end_of_step*, *release* and *transaction*. Without the protected object semantics, is is possible that the other philosophers do not progress (they all remain in state *InQueue*). Thus, the philosopher 1 goes to state *InQueue* and removes the last token of place *NotInQueue*: the siphon is empty and the net is deadlocked. **This is the reason why the implementation of the server with a task is not deadlock free**. Indeed, if just after having serviced the entry release for the philosopher 0, the server task is preempted (it loses the processor due to the scheduler or due to a call to a blocking action like a delay), then the only eligible task is the philosopher 0, that can call the entry $request(0)$ and then is put in the queue of this entry. When the server becomes active again, all the entries $request(x)$ are then closed, and the program is deadlocked.

However, the Ada95 processing policy of protected objects ensures that at the end of a protected call, already queued entries (whose barriers are true) take precedence over new calls. So, when the philosopher 0 fires *release*, it enters state *Thinking* and the philosopher 1 enters state *Eating* marking anew the place *NotInQueue*. The siphon $D = \{NotInQueue\}$ is then controlled as a result of the Ada95 processing policy of protected objects and then the model is really deadlock free.

With the Abort Part

In we consider the complete model (including the abort part), then place *NotInQueue* is not a siphon, and previous complications cannot occur. Indeed, when a philosopher x is in state *InQueue*, transition *control.stop_me* is always fireable, and its firing marks place *InQueue* again. Therefore, a circular deadlock cannot occur.

4.3.2 Proving Starvation Freeness of the Model

Without the Abort Part

The philosophers can be in six different states, but only one (the state *InQueue*) is a state in which philosophers may remain indefinitely due to a bad policy of the server. We have to prove that any philosopher necessarily leaves the state *InQueue* after a finite time.

The key point in proving the starvation prevention of this solution is based on the fact that when a philosopher x goes in state *InQueue*, it takes the resource $NotInQueue(x)$ and forces the philosopher $x + 1$ to wait until x leaves the state *InQueue* for the state *Eating*.

Let us suppose that at a marking M, a philosopher x in state *InQueue* ($M(InQueue)(x) = 1$) cannot fire the transition *requestQ* and access the state *Eating*. There are four cases :

1. $M(Eating(x-1)) = 1$ and $M(Eating(x+1)) = 1$

   After a finite time (the net is deadlock free), $x-1$ or $x+1$ will fire transition *release*, and we fall then under the next cases.

2. $M(Eating(x+1)) = 1$ and $M(Eating(x-1)) = 0$

   When the philosopher $x+1$ fires transition *release*, it enters state *Thinking* and we fall under the next cases.

3. $M(Eating(x+1)) = 0$ and $M(Eating(x-1)) = 1$

   When the philosopher $x-1$ fires transition *release* it enters state *Thinking* and, as x is in the queue, $x+1$ is still not in state *Eating*, and we fall under the next case.

4. $M(Eating(x+1)) = 0$ and $M(Eating(x-1)) = 0$

   At such a marking we have to consider two sub-cases :

   (a) $M(InQueue(x-1)) = 0$: $x+1$ cannot enter state *Eating* because $M(NotInQueue(x)) = 0$ (x is in the queue) and $x-1$ cannot enter state *Eating* or enter state *InQueue* because preference is given to already queued tasks (here x).

   So the philosopher x necessarily fires *requestQ* before $x-1$ or $x+1$ can move and x accesses to the state *Eating*.

   (b) $M(InQueue(x-1) = 1)$: if $x-1$ can access the state *Eating*, when it fires *Release* we fall under the previous sub-case.

   Otherwise, the only possibility is that $x-1$ is waiting that $x-2$ fires *requestQ*, which is waiting that $x-3$ fires *requestQ*, and so on; in this case, the net is deadlocked, and this is impossible as proved above. So, x fires necessarily *requestQ* after a finite time.

With the Abort Part
If we consider the model including the abort part, the model does not prevent starvation. Indeed, we cannot forbid a philosopher to fix too short a deadline putting itself into self-starvation. This problem is not due to the resource allocation policy but is due to a possible bad strategy of the philosophers.

5 Discussion

So far we have considered a solution where all the resources are requested once and are allocated globally. If a client cannot be given all the requested resources (i.e., two chopsticks in the case of the philosophers example), it is postponed until these resources all become available; no resources are set apart for its postponed claim; all the unallocated resources are available for any other request.

It is a well known result that such a policy avoids deadlock, but does not prevent starvation.

5.1 A Deadlock Free Solution Which can Lead to Starvation

As an example, consider the dining philosophers solution proposed by B.Brosgol in [Bro96].

```
-----------------------------------------------------
with common; use common;
package Brosgol_Allocation_chops is
   procedure request(me : in philo_id);
   procedure release(me : in philo_id);
private
   type Boolean_Array is array(philo_id) of boolean;
end Brosgol_Allocation_chops ;

package body Brosgol_Allocation_chops is
   protected server is
      entry get_pair(X : in philo_id);
      procedure release_pair(x : in philo_id);
   private
      available : boolean_array := (others => true);
      entry please_get_pair(x : in philo_id);
      Flush_Count: Natural := 0;
   end server;

   procedure request(me : in philo_id) is
   begin server.get_pair(me); end request;

   procedure release(me : in philo_id) is
   begin server.release_pair(me ); end release;

   protected body server is
      entry get_pair(x :in philo_id) when true is
      begin
         if available(x) and available(x + 1)then
            available(x) := false;
            available(x + 1) := false;
         else
            requeue please_get_pair with abort ;
         end if;
      end get_pair;

      entry please_get_pair(x :in philo_id)
      when flush_count > 0 is
      begin
         flush_count := flush_count - 1;
         if available(x) and available(x + 1) then
            available(x) := false;
            available(x + 1) := false;
         else
            requeue please_get_pair with abort ;
         end if;
      end please_get_pair;

      procedure release_pair(x : in philo_id) is
      begin
         available(x) := true;
```

```
        available(x + 1) := true;
        flush_count := please_get_pair'count;
     end release_pair;
  end server;
end Brosgol_Allocation_chops ;
--------------------------------------------------------
```

Consider now the following execution with 5 philosophers: 0 calls get_pair, gets the chopsticks and eats. 1 calls get_pair and is requeued to please_get_pair. 2 calls get_pair, gets the chopsticks and eats. 3 calls get_pair and is requeued to please_get_pair. 4 calls get_pair and is requeued to please_get_pair.

At this state, 0 and 2 are eating while 1, 3 and 4 are in the queue of please_get_pair in this order (it is a FIFO queue).

Let us suppose that 2 releases its chopsticks. Because of the semantics of the protected object, as the guard of please_get_pair has become true, the code of this entry is re-executed first for the philosopher 1, then for 3 and then for 4. 1 cannot be served because 0 is still eating and is then requeued, 3 is served and 4 is then also requeued. At this point, 0 and 3 are eating while 1 and 4 are in the queue of please_get_pair. Let us suppose now that 3 releases its chopsticks. For the same reasons, the requests of 1 and 4 are re-examined but cannot be satisfied. If 2 then calls get_pair, it is served. If now 0 releases its chopsticks, 4 can eat but 1 remains in the queue. And so on. As long as his two neighbors eat in alternation such that at least one of them is always eating, there is no possibility for philosopher 1 to get his pair of chopsticks, even if all the other philosophers release their chopsticks regularly.

5.2 The Approach Used to Avoid Starvation

The idea is to use a kind of control such that, if a client x requests resources, all the other clients cannot continue to be served indefinitely before x. A function is added which forces an order of allocation once x is blocked. The key problem is to be sure that this additional function does not introduce a deadlock when several clients are blocked.

This additional function is stated in terms of additional resources. Thus it can be tractable in Petri net theory. This leads to the following presentation.

Each client x possesses a personal resource $R(x)$ which can be allocated to x only. This resource $R(x)$ is managed in the following way:

- initially $R(x)$ is free for all x
- when x claims of shared resources (chopsticks in the example of philosophers) and is postponed, then $R(x)$ is consumed by x
- when x is given shared resources, $R(x)$ becomes free again. Moreover, we impose the condition that there exists one and only one client, denoted $A(x)$, such that $A(x)$ cannot be served if $R(x)$ is not free.

We also require also that $A(x) \neq A(x')$ if $x \neq x'$. This means that $A(x)$ is a permutation in the set of clients. We also require that for all x : [3]

- $\forall z \in 1..N-1, A^z(x) \neq x$ and that
- $A^N(x) = x$

If no deadlock results from the introduction of this blocking policy using $A(x)$ and $R(x)$, then this policy avoids starvation. To prove it, let us suppose that a client x makes a claim and is never served. When x is postponed for the first time, it consumes $R(x)$. Thus the client $A(x)$ is prevented from being served. $A(x)$ also consumes its personal resource $R(A(x))$, and this blocks the client $A(A(x))$. This also blocks the client $A^3(x))$ and so on. Because we require that $\forall z \in 1..N, A^z(x) \neq x$ and that $A^N(x) = x$, if x is never served, it generates a chain of blocking which corresponds to a global deadlock, which contradicts our initial supposition.

Let us use this approach to the Brosgol implementation presented above and add a variable named $Requestor(x)$ which denotes that the philosopher x is being postponed. $Requestor(x)$ corresponds to the resource $R(x)$ and the permutation $A(x)$ is $A(x) = (x+1)\,modulo\,N$.

```
--------------------------------------------------------
package body Fair_Brosgol_Allocation_chops is

   protected server is
      entry get_pair(X : in philo_id);
      procedure release_pair(x : in philo_id);
   private
      available : boolean_array := (others => true);
                                            -- chopsticks
      requestor : boolean_array := (others => false);
                                            -- philosophers
      entry please_get_pair(x : in philo_id);
      Flush_Count: Natural := 0;
   end server;

   procedure request(me : in philo_id) is
   begin server.get_pair(me); end request;

   procedure release(me : in philo_id) is
   begin server.release_pair(me ); end release;

   protected body server is

      entry get_pair(x :in philo_id) when true is
      begin
         if available(x) and available(x + 1) and
            not requestor(x-1) then
            available(x) := false;
            available(x + 1) := false;
         else
            requestor(x) := true;
            requeue please_get_pair;
         end if;
      end get_pair;
```

[3] $A^z(x)$ denotes $A(A(...(A(x))))$ z times, and $A^0(x) = x$

```
      entry please_get_pair(x :in philo_id)
      when flush_count > 0 is
      begin
         flush_count := flush_count - 1;
         if available(x) and available(x + 1) and
            not requestor(x-1) then
            available(x) := false;
            available(x + 1) := false;
            requestor(x) := false;
         else
            requeue please_get_pair;
         end if;
      end please_get_pair;

      procedure release_pair(x : in philo_id) is
      begin
         available(x) := true;
         available(x + 1) := true;
         flush_count := please_get_pair'count;
      end release_pair;
  end server;
end Fair_Brosgol_Allocation_chops;
--------------------------------------------------------
```

This solution is very close to the solution presented in Sections 2 and 3 (where $R(x)$ is consumed when x is queued at the request entry). The same kind of proofs allows us to state that this solution is deadlock free (and then also starvation free) when the allocator is implemented with a protected object and that, on the contrary, this solution is not deadlock free when the allocator is implemented with a server task.

Remark :

1. The Brosgol implementation supposes that the queues are served FIFO. If there are not, it is necessary to use two queues in alternation and two private entries [BW95].
2. If we also want to take care of abort situations (and use the requeue with abort clause), it is necessary to note the state of the philosophers, because a philosopher can be aborted when being postponed (he is not eating and has no chopsticks to release) or when eating (he has two chopsticks to release).
3. The initial Brosgol implementation can also be transformed in the following way to become fair :
   (a) all requests are put in a unique FIFO queue (for example, the entry `get_pair` just requeues all claims at the entry `please_get_pair`) and
   (b) no request from this queue can be serviced before the first request. Here the fairness relies entirely on the FIFO service of the requests.

5.3 Comparing the Controlling Capabilities of Protected Objects and of Server Tasks

The following points are to be considered:

1. About barriers and guards

   - The barriers of protected object entries and the guards of tasks entries are evaluated one after the other in mutual exclusion.

   - The barriers and guards both can be programmed with encapsulated data that are private to the protected object or to the task. The modifications to these private data are also executed in mutual exclusion with the evaluation of barriers and guards.

   - Therefore, the analysis of the parameters of a request, which may result in postponing and requeuing the request, can be recorded in a variable (such as the number of already postponed requests). This variable can be used in the expression of barriers or guards in order to prevent a circular blocking.

2. About the *count* attribute

   - The *count* attribute of a protected entry is computed in a mutually exclusive region. Indeed, a calling task is counted as blocked at an entry only once the corresponding barrier has been evaluated to false. Similarly, mutual exclusion is used to handle the abort action which may decrease the *count* attribute.

   - On the other hand, modifications of the *count* attribute of task entries may occur concurrently at any moment and not only during the evaluation of guards. The *count* attribute of a task entry is increased at the arrival of a new call unless the called task is at the rendez-vous. This attribute is decreased either when the caller is serviced or as a result of a timed entry call or of an abort. Thus, between two successive evaluations of guards, several *count* attributes can change and cause several guards to become false.

   The evolution of this attribute is difficult to predict and control and may cause circular blocking.

3. About the standard service policy

   - In a protected object, internal waiting tasks take precedence over external tasks.

   - In a server task, all open entries (even the private ones) are equivalent.

From this brief summary, let us remark that, conceptually, the behavior of protected objects that do not use the *count* attribute can be simulated by a server task. This simulation may use additional private variables and requeue statement, and it may modify the guard expressions.

For example, the previous solution given is section 5.2 can be adapted. To provide a fair server task, the entry `please_get_pair` is given preference over the entry `get_pair` by the addition of the guard `flush_count = 0`.

```
--------------------------------------------------------
-- reliable task server with requeue with abort
--------------------------------------------------------
```

```
with  common;  use  common;
package reliable_and_fair_server_chops is
   procedure request(me : in philo_id);
   procedure release(me : in philo_id);
end reliable_and_fair_server_chops ;

package body reliable_and_fair_server_chops is
   type boolean_array is array(philo_id) of boolean ;
   task server is
      entry get_pair(x :in philo_id);
      entry release_pair(x : in philo_id);
   private
     entry please_get_pair(x : in philo_id);
   end server;
   procedure request(me : in philo_id) is
   begin
      server.get_pair(me);
   end request;
   procedure release(me : in philo_id) is
   begin
      server.release_pair(me );
   end release;
   task body server is
      flush_count : integer := 0;
      available : boolean_array := (others => true);
      --available denotes availability of the chopsticks
      requestor : boolean_array := (others => false);
      eating : boolean_array := (others => false);
      -- requestor and eating denote philosopher states
   begin
      loop
         select
            when flush_count = 0 =>
               accept get_pair(x : in philo_id) do
                  if available(x) and
                    not requestor(x - 1) and
                    available(x + 1) then
                     available(x) := false;
                     available(x + 1) := false;
                     eating(x) := true;
                     -- useful to know if aborted
                  else
                     requestor(x) := true;
                     requeue please_get_pair with abort;
                  end if;
               end get_pair;
         or
            when flush_count > 0 =>
               accept please_get_pair(x: in philo_id) do
                  flush_count := flush_count - 1;
                  if available(x) and
                    not requestor(x - 1) and
                    available(x + 1) then
                     available(x) := false;
                     available(x + 1) := false;
                     requestor(x) := false;
                     eating(x) := true;
                     -- useful to know if aborted
                  else
                     requeue please_get_pair with abort;
                  end if;
               end please_get_pair;
         or
            accept release_pair(x : in philo_id) do
               if eating(x) then
                  available(x) := true;
                  available(x + 1) := true;
                  eating(x) := false;
               end if;
               requestor(x) := false;
               -- can be done anyway
               flush_count := please_get_pair'count;
            end release_pair;
         or
            terminate;
         end select;
      end loop;
   end server ;
end reliable_and_fair_server_chops ;
--------------------------------------------------------
```

Remark : As the number of tasks in the queue of the task entry `please_get_pair` may be different from the value given by `please_get_pair'count`, the exact solution should be implemented with two queues and two entries used in alternation. We have given the present solution for the sake of simplicity.

5.4 Another Policy

A generalization of the policy given above is to allow the allocator to reserve the chopsticks one by one. The allocator still postpones the requesting philosophers until both chopsticks are available. This policy is reliable and fair if :

1. a postponed philosopher cannot be overtaken by an infinitely rapid neighbor that releases its chopsticks and requests them anew, and also if

2. the blocking due to the reservation of one chopstick does not lead to a deadlock.

The eggshell model of the protected object provides these two conditions and allows the following solution to be reliable and fair, while again the server task implementing the same solution fails to provide both deadlock freeness and fairness.

```
--------------------------------------------------------
with  common;  use  common;
package chopsticks_object is
   procedure request(me : in philo_id);
   procedure release(me : in philo_id);
end chopsticks_object ;  -- end of package declarations

package body chopsticks_object is
   type boolean_array is array(philo_id) of boolean ;

   protected chopsticks is
      entry get_pair(philo_id);
      procedure release_pair(x : in philo_id);
   private
      available : boolean_array := (others => true);
      entry finish_pair(philo_id);
   end chopsticks ;
   procedure request(me : in philo_id) is
   begin
      chopsticks.get_pair(me);
   end request;

   procedure release(me : in philo_id) is
   begin
      chopsticks.release_pair(me);
   end release;

   protected body chopsticks is
      entry get_pair(for i in philo_id) when
        available(i) is
      begin
         available(i) := false;
  requeue finish_pair(i + 1);
      end get_pair;

      entry finish_pair(for i in philo_id) when
        available(i) is
      begin
         available(i) := false;
      end finish_pair;

      procedure release_pair(x : in philo_id) is
      begin
         available(x) := true;
         available(x + 1) := true;
      end release_pair;
   end chopsticks ;
end chopsticks_object ;  -- end of package body
--------------------------------------------------------
```

Remark :

1. If we want to take care of abort situations (and use the requeue with abort clause), it is necessary to note the state of the philosophers because a philosopher i can be aborted when being postponed at the entry `get_pair` (it is not eating and therefore has no chopsticks to release) or when being postponed at the entry `finish_pair` (it is not eating and then has its chopstick i to release) or when eating (it has then two chopsticks, i and $i+1$, to release).
2. A correct server task can be implemented with an additional variable

   `requestor : boolean_array := (others => False)`

   The guard of the entry `get_pair(i)` is now

   `available(i) and not requestor(i)`

   The code of the entry `get_pair(i)` sets `requestor(i+1)` to `True` while the code of the entry `finsih_pair(i)` resets `requestor(i)` to `False`.

6 Conclusion

It has been shown that programming resource allocation with Ada95 can lead to programs which are easy to design and to understand and which can be proven to be (or not to be) deadlock free and starvation free.

From the case study presented in this paper, some general conclusions can be drawn.

A policy aiming at preventing deadlock, i.e. at preventing circular wait of requested resources, may introduce starvation. Similarly a policy aiming at preventing starvation may introduce deadlock.

The eggshell model of Ada 95 protected objects, which queues new requests sequentially (and therefore provides a reliable attribute count) and which serves, in priority order, internally waiting tasks before external tasks, is a very powerful tool and is basic for proving the absence of starvation. Especially, when a task releases a resource while the request of another task has been postponed inside the protected object, then this latter task will always be serviced before any new request of the former task. This prevents an infinitely fast task from monopolizing the resource. This behavior is the basic reason why the solutions presented in this paper are safe and fair when implemented with protected objects and not when implemented with a server task.

The Ada95 schema relies on automaton-like and state transition execution. This is exactly what is used when modeling a problem with Petri nets (or with some state transition model). This is why we were able to program the allocation policies and jointly to validate (or invalidate) the chosen implementation. Other examples of this approach are given in [BKPP97] and [KPP97]. Thus this cooperative approach may be a basis for systematic construction of reliable concurrent programs.

Acknowledgments

The authors thank the anonymous referees for their valuable comments and are grateful to Norman Cohen and to Christine Ausnit-Hood for helping us to improve the English of this paper.

References

[Ber86] G. Berthelot. Transformations and decompositions of nets. In *Advances in Petri Nets*, number 254 in LNCS, pages 359–376. Springer-Verlag, 1986.

[BHPP97] F. Breant, S. Haddad, and J.F. Pradat-Peyre. Characterizing new reductions by means of language and invariant properties. Technical Report 97-04, Conservatoire National des Arts et Métiers, laboratoire Cedric, "ftp.cnam.fr/ pub/ CNAM/ cedric/ tech_reports/", 1997.

[BKPP97] K. Barkaoui, C. Kaiser, and J.F. Pradat-Peyre. Petri nets based proofs of Ada 95 solution for preference
control. Technical Report 97-08, Conservatoire National des Arts et Métiers, laboratoire Cedric, "ftp.cnam.fr/pub/CNAM/cedric/tech_reports/", 1997.

[BPP96] K. Barkaoui and J.F. Pradat-Peyre. On liveness on controlled siphons in Petri nets. In Reisig, editor, *Petri Nets, Theory and Application*, number 1091 in LNCS. Springer-Verlag, 1996.

[Bro96] B. Brosgol. The dining philosophers in Ada95. In *Reliable Software Technologies-Ada-Europe'96*, number 1088 in LNCS, pages 247–261. Springer-Verlag, 1996.

[BW95] A. Burns and A. Wellings. *Concurrency in Ada*, chapter 6.11, pages 134–137. Cambridge University Press, 1995.

[CG77] P.J. Courtois and J. Georges. On starvation prevention. In *RAIRO Informatique/Computer Science*, volume 11, 1977.

[Dij71] E.W. Dijkstra. Hierarchical ordering of sequential processes. In *Acta Informatica*, number 1, pages 115–138, 1971.

[Had91] S. Haddad. A reduction theory for colored nets. In Jensen and Rozenberg, editors, *High-level Petri Nets, Theory and Application*, LNCS, pages 399–425. Springer-Verlag, 1991.

[Int95] Intermetrics Inc. *Ada 95 Rationale*, 1995.

[KPP97] C. Kaiser and J.F. Pradat-Peyre. Reliable resources allocation with Ada95 preference control. Technical Report 97-06, Conservatoire National des Arts et Métiers, laboratoire Cedric, "ftp.cnam.fr/pub/CNAM/cedric/tech_reports/", 1997.

[MSS89] T. Murata, B. Shenker, and S.M. Shatz. Detection of Ada static deadlocks using Petri nets invariants. *IEEE Transactions on Software Engineering*, Vol. 15(No. 3):314–326, March 1989.

[Rei83] W. Reisig. *EATCS-An Introduction to Petri Nets*. Springer-Verlag, 1983.

[TSM90] S. Tu, S.M. Shatz, and T. Murata. Applying Petri nets reduction to support Ada-tasking deadlock detection. In *Proceedings of the 10th IEEE Int. Conf. on Distributed Computing Systems*, pages 96–102, Paris, France, June 1990.

A Appendix

A.1 The Package Body SCENE

```
-------------------------------------------------------
with Ada.Text_IO, Ada.Numerics.Float_Random,
  Ada.Numerics.Discrete_Random;
use Ada.Text_IO, Ada.Numerics.Float_Random;
package body scene is
   G1 : Generator;
   -- uniform distribution from 0.0 to 1.0

   procedure thinking(x : in philo_id) is
      how_long : Float range 0.0 .. 1.0;
   begin
      how_long := 0.1 * Random(G1);
      -- we suppose that Random executes
      -- in mutual exclusion
      put_line("the philosopher : " &
               philo_id'image(x) &
               " is            thinking");
      delay(duration(how_long)); -- this takes a while
      put_line("the philosopher : " &
               philo_id'image(x) &
               " has              finished thinking");
   end thinking;

   procedure eating(x : in philo_id) is
      how_long : Float range 0.0 .. 1.0;
   begin
      how_long := 0.1 * Random(G1);
      put_line("the philosopher  " &
               philo_id'image(x) &
               " is                        eating");
      delay (Duration(How_Long));
      put_line("the philosopher  " &
               philo_id'image(x) &
               " has               finished eating");
   end eating;

   function random_duration (d1,d2 : in duration)
                                return duration is
      x : Float range 0.0 .. 1.0;
   begin
      x := Random(G1); -- random execution duration
      return duration(Float(D1) + Float(d2 - d1) * x);
   end random_duration ;

end scene;
-------------------------------------------------------
```

A.2 The Complete Model Corresponding to the Program

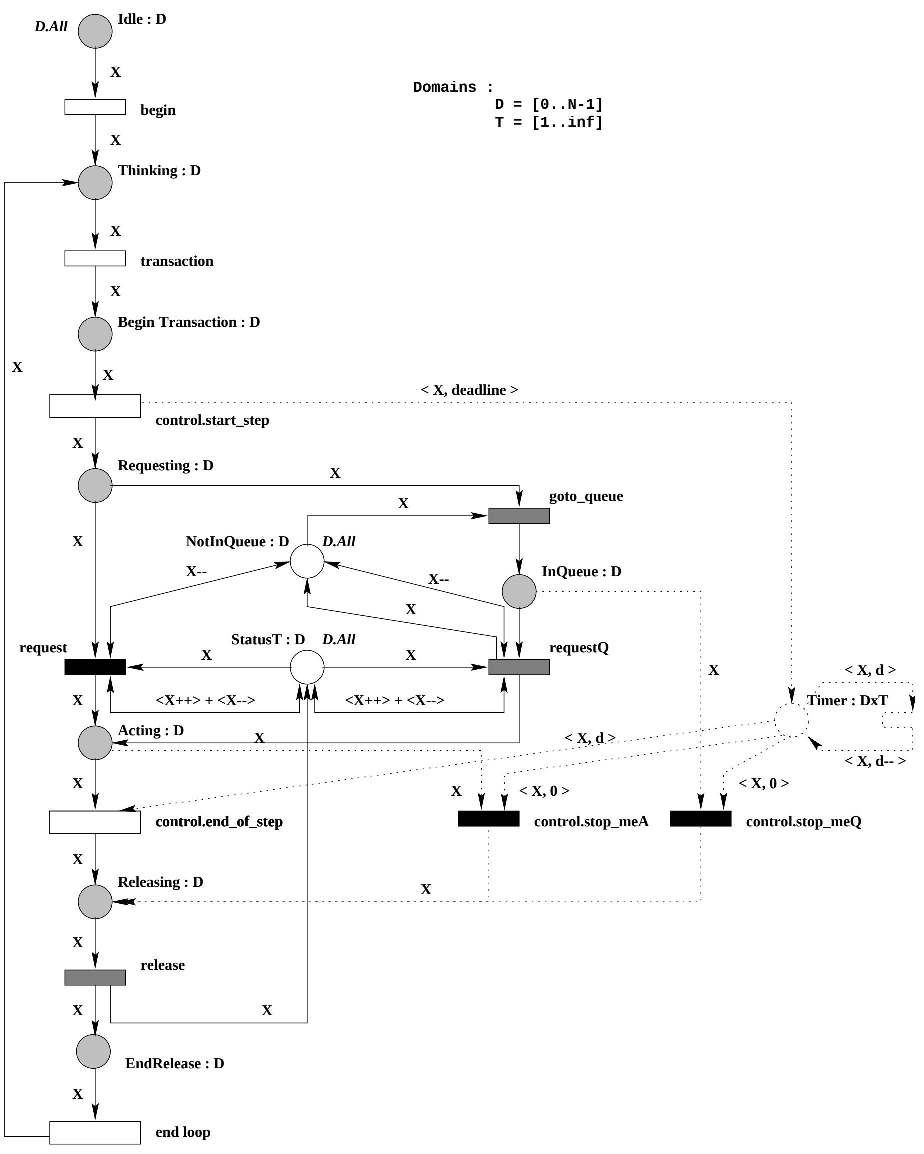

Figure 5: The dining philosophers