Petri Nets Based Proofs of ADA 95 solution for Preference Control

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

Abstract

This paper presents correctness proofs of Ada 95 preference control solution for the dining philosophers paradigm. Preference control is the ability to satisfy a request depending on the parameters passed in by the calling task and often also on the internal state of the server. In Ada 95, this schema is implemented with protected objects, entry families and requeue statements within the protected object.. The aim of our presentation is to show that the preference control can be described in terms of states and transitions, similar to reactive automaton descriptions. This description, which can be done in terms of colored Petri nets, can lead jointly to validate the chosen implementation and to program it with protected objects and requeue statements of Ada95. The paper is issued from an examination exercise for our students that have followed a course on Programming and Validation of Concurrent Applications and is presented in form of progressive steps for which the students were expected to give answer. This is why the paradigm of dining philosophers was chosen. The paper contains three programmed solutions and proofs of absence of deadlock and of starvation.

1. Introduction

1.1. Preference control and Ada95

Preference control is the ability to satisfy a request depending on the parameters passed in by the calling task and often also on the internal state of the server. The paradigm of the dining philosophers is a good example where a client task requests a particular number of different resources, where some resources might be available, while others are in use, where a parameter indicates the client specific request, and where the server is controlling the allocation in order to guarantee some correctness properties (absence of deadlock and absence of starvation).

Many papers discussing preference control have appeared in the literature and have shown that there was no good way to implement it in Ada 83([Elr88], [WKT84], [BLW87]). One way required exporting several entries that had to be called in a particular order; this violated information-hiding principles, and caused race conditions, because events could occur between the multiple calls. Another way required an extra agent task which was created dynamically in order to handle the request. Other solutions required the requesting task to poll the server.

In Ada95 [Int95a, Int95b], the tasking of Ada 83 has been enhanced with additions such as protected types and the requeue statement. The requeue statement has been added to provide preference control and thereby overcome race conditions which could arise in Ada 83. In Ada 83, all solutions had to share preference control decisions amongst the client and the server and thus required some form of distributed control. In Ada 95, all the decisions rely on information stored or accessible in the server context. This locality of information and of decision allows both much simpler implementations and much simpler proofs.

1.2. Colored Petri nets

The technique used for proving correctness is based on Petri nets theory.

When the solution is modeled by a colored Petri net, several methods can be used to prove some properties on the model : the reduction theory that allows us to define an equivalent model on which properties are more easy to prove [Had91, Ber86], the places invariants of the net that define some invariant properties on reachable marking [CHP91], necessary and sufficient conditions of liveness based on the notion of controlled siphons [BPP96b] and the construction of the symbolic reachable marking graph that provides a

compact description of all possible states of the model [CDFH90]. All these methods have been implemented in tools, such as GreatSPN, CPN-AMI or DesignCPN which allow us to make automatic symbolic and/or parameterized proofs.

1.3. Progressive design of a reliable solution to the dining philosophers problem

The aim of our presentation is to show that the concurrent programming construct which is called preference control, and which is necessary to implement complex servers, can be described in terms of states and transitions, similar to reactive automaton descriptions. This description, which can be done in terms of Petri nets, can lead jointly to validate the chosen implementation and to program it with protected objects and requeue statements. This could be the basis of a systematic method of reliable concurrent programming with Ada 95 when implementing preference control.

We present some well known solutions to the dining philosophers problem and give Petri nets proofs of their implementation in Ada 95. This was the core of an examination exercise for students that have followed a course on Programming and Validation of Concurrent Applications [Kai96, BPP96a, Flo96]. The aim of this presentation is to introduce a systematic method of programming and validation, which gracefully leads to a reliable implementation with protected types and requeue.

In section 2, we give a first approach where the client requests the resources all at once (i.e. the two specific chopsticks) and waits for them since it cannot proceed with a subset of the requested resources (i.e. the philosopher cannot eat with only one chopstick, or cannot do another activity when he is hungry); the server allocates the request resources globally when they are available. This is a well-known solution for deadlock prevention; however it does not prevent starvation.

The implementation in Ada95 uses a protected object with an entry family [Bro96]. The reduction theory of Petri nets is used to prove that this implementation is deadlock-free.

This solution was chosen to show a style of concurrency respecting a strong separation of resource management and control between client and server; this respects the information-hiding principle. The client has no idea of how the server proceeds and has not to know it to call the service.

In section 3, another well known solution is presented. An additional constraint (the dining philosophers must be seated before they are allowed to request their first chopstick and all philosophers cannot have chairs) is introduced to prevent the resource allocation method to reach a state where a deadlock situation holds. In this implementation, the client has to know the solution chosen by the server (this remains true if the sequence of call is embedded in a procedure or in an agent task). This solution prevents deadlock and starvation. This is proven by using places invariants of the net (positive flows and siphons).

In section 4, we present a solution which combines both advantages : it respects the information-hiding principle and it provides a clever and safe solution (the safety does not rely on a specific queuing policy as in [Bro96]). The safety of the corresponding Petri net is proven using reduction theory, places invariants of the net and a new construction of the symbolic reachability graph. This method can be extended to other solutions of the dining philosophers problem which can be implemented in a server with the same client interface. Another method of deadlock prevention, called linear ordering classes of resources and servicing the resources according to this order, can be used and provides two other known solutions. In one solution, chopsticks are ordered according to the increasing value of their index and fork X is requested first only when (X+ 1) mod N is greater than X, otherwise fork (X + 1) mod N is required first. In another solution, one class of resources contains odd numbered chopsticks while the other class contains the even numbered ones.

The continuation part of the paper is now presented in form of progressive steps for which the students were expected to give answer.

2. Approach with information hiding and deadlock prevention

We have studied during the course different solutions to the dining philosophers problem regarding resources allocation [Kai96, BPP96a]. As recall 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 only N/2 philosophers can eat simultaneously). Design a solution so that for an arbitrary integer M, each philoso-

pher is guaranteed to perform Eat-Think sequence (at least) M times."

2.1. Defining briefly Petri nets

A Petri net [Rei83] is a 4-tuple $\langle P, T, W^+, W^- \rangle$ 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): ${}^\bullet p = \{t \in T | W^+(p,t) > 0\}$ and $p^\bullet = \{t \in T | W^-(p,t) > 0\}$.

2.2. Recalling basic properties of Petri nets

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

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

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

2.3. Explaining differences between Petri nets and colored nets

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 2.1 *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* $Bag((C(p))$*.* [1] *The incidence matrix of the net is defined by* $W = W^+ - W^-$*.*
- M_0 *is the initial marking of the net and is an application defined on* P *with* $M_0(p) \in Bag(C(p))$*.*

Definition 2.2 *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,* $M[t >_{c_t} M'$*.*

[1] 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

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 most usual color functions for a class C are the identity (or selection) denoted X_C (or Y_C, Z_C, ...), the diffusion or the global synchronization over the class C, denoted All_C, and the successor mapping denoted X_C++.

2.4. Algorithm and colored net model of the first solution

The client requests the resources all at once (i.e. its chopstick and that of its right-hand side neighbor) and waits for them since it cannot proceed with a subset of the requested resources (i.e. the philosopher cannot eat with only one chopstick, or cannot do another activity when he is hungry); the server allocates them globally when they are available.

In Ada 95, a protected object encapsulates and provides synchronized shared access to the private data of that object. 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. Entry families specify a name for an entry family index which can be used in the entry body allowing to describe the entry bodies of the whole family in a unique text.

Using Ada95 protected object and entry families, the first solution is implemented as follow (Ada program 1).

```
procedure le_premier_repas is
    N : constant integer := 5;
    type id is mod N;
    type tab is array(id) of boolean;

    protected repas is
        entry demander(id); -- entry family
        procedure conclure(X : in id);
        private -- the chopsticks
            Baguettes : tab := (others => true);
    end repas;

    protected numero is
        procedure unique(resultat: out id);
        private compte : id := id'first;
    end numero;
    task type philo;
    philosophe: array (id) of philo;
    task body philo is ego : id;
    begin
        numero.unique(ego);
        loop
            pense(ego); -- think
            repas.demander(ego);
            mange(ego);  -- eat
            repas.conclure(ego);
        end loop;
    end philo;
    protected body repas is separate;
    protected body numero is separate;
begin null;
end le_premier_repas;

-------------------------------------------------------------
separate(le_premier_repas)
protected body numero is
    procedure unique(resultat : out id) is
    begin
        resultat := compte;
        compte := compte+1; -- addition modulo N
    end unique;
end numero;

-------------------------------------------------------------
separate(le_premier_repas)
protected body repas is
    entry demander(for i in id)
        when Baguettes(i) and Baguettes(i+1) is
    begin
        Baguettes(i):= false;
        Baguettes(i+1):=false;
    end demander;

    procedure conclure(X: in id) is
    begin
        Baguettes(X) := true;
        Baguettes(X+1):=true;
    end conclure;
end repas;
```

Ada program 1 : approach with information hiding and deadlock prevention

This implementation is modeled by the following colored net (fig. 1).

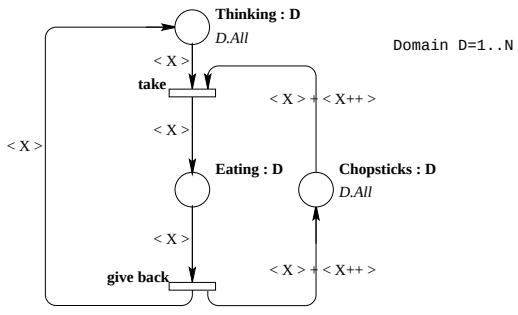

Figure 1. Colored net of the Ada program 1

2.5. Interpreting this colored net

This net is composed of three places ($Thinking$, $Eating$ and $Chopsticks$) and of two transitions ($take$ and $give\,back$). Places $Thinking$ and $Eating$ model the two different visible states of the N philosophers. For instance, a philosopher x is in the state eating when a token of color x is in the place $Eating$. Place $Chopsticks$ models the available chopsticks.

Initially, all philosophers are in state thinking and all chopsticks are free. This is modeled by the definition of the initial marking $M_0(Thinking) = D.All, M_0(Eating) = 0, M_0(Chopsticks) = D.All$.

Transition $take$ can be fired for a color x when place $Thinking$ contains at least one token of color x and when place $Chopsticks$ contains at least one token of color x and one token of color $x+1\,mod\,N$. Upon firing, the needed tokens are removed and a token of color x is added to place $Eating$ This transition models the entry `demander`.

Transition $give\,back$ models the procedure `conclure` and is fireable when place $Eating$ contains at least one token of color x (the philosopher x is in state eating). This firing produces the two tokens x and $x++$ in place $Chopsticks$ (the chopsticks x and $x+1\,mod\,N$ become free again) and one token x in place $Thinking$ (the philosopher x is now in state thinking).

2.6. Using the reductions theory of Petri nets to prove that this solution is deadlock free

Petri net (or colored net) reductions allow one to simplify a model while preserving some properties of the initial model [Ber86, Had91] A reduction is characterized by three definitions : the application conditions, the reduced net, and the properties that are equivalent to check either in the original net or in the reduced one.

In this net one can apply the agglomeration of transition $take$ with transition $give\,back$. By definitions of these reductions, proving that the reduced net is deadlock free proves that the original one is also deadlock free.

The reduced net is obviously live since the only transition is a "test" transition (its firing preserves the number and the color of tokens contains in each place) and is initially fireable for each instance of D by definition of the initial marking. [2]

[2] the two places of the net can be suppressed by an other reduction : suppression of implicit place [Had91]

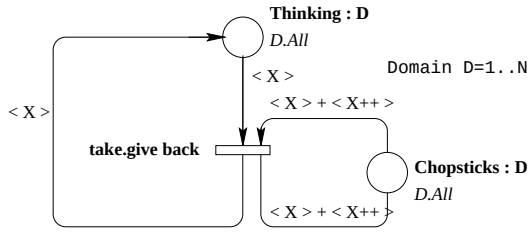

Figure 2. The equivalent net after the agglomeration of transition $take$ with $giveback$

3. Approach with deadlock and starvation prevention

3.1. Algorithm and colored model of the second solution

The second solution to our problem is based on a step by step allocation of the resources. A philosopher x that wants to eat begins by taking a seat, then he takes the chopstick numbered x and then the chopstick numbered $x+1$. At each resource request, he has to wait until this resource becomes free.

```
procedure le_deuxieme_repas is
    N : constant integer := 5;
    type id is mod N;

    protected chaise is
        entry prendre;
        procedure rendre;
        private libre : integer := N-1;
    end chaise;

    protected type ressource is
        entry prendre;
        entry rendre;
        private pris : boolean:=false;
    end ressource;

    protected numero is
        procedure unique(resultat: out id);
        private compte : id := id'first;
    end numero;

    task type philo;
    philosophe: array(id) of philo;
    baguettes : array(id) of ressource;

    task body philo is
    ego : id;
    begin
        numero.unique(ego);
        loop
            pense(ego);
            chaise.prendre;
            baguette(ego).prendre;
            baguette(ego + 1).prendre;
            mange(ego);
            baguette(ego).rendre;
            baguette(ego + 1).rendre;
            chaise.rendre;
```

```
        end loop;
    end philo;
    protected body chaise is separate;
    protected body ressource is separate;
    protected body numero is separate;
begin null;
end le_deuxieme_repas ;

-------------------------------------------------------------
separate(le_deuxieme_repas )
protected body chaise is
    entry prendre when libre > 0 is
    begin libre := libre - 1; end prendre;
    procedure rendre is
    begin libre := libre + 1; end rendre;
end chaise;

-------------------------------------------------------------
separate(le_deuxieme_repas )
protected body ressource is
    entry prendre when not pris is
    begin pris := true; end prendre;
    entry rendre when pris is
    begin pris := false; end rendre;
end ressource;

-------------------------------------------------------------
separate(le_deuxieme_repas)
protected body numero is
    procedure unique(resultat : out id) is
    begin
        resultat := compte;
        compte := compte+1; -- addition modulo N
    end unique;
end numero;
```

Ada program 2 : Ada95 program for safe allocation

The following net (fig. 3) models this behavior.

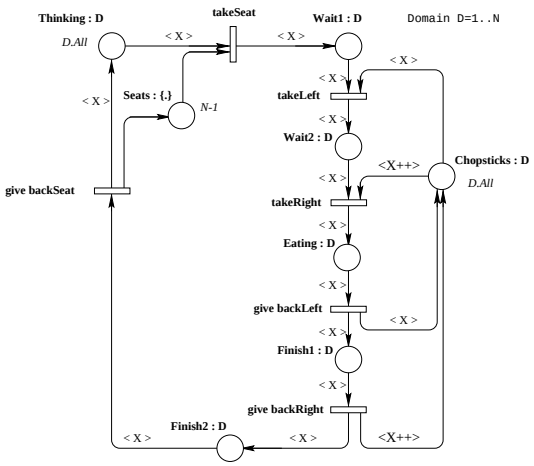

Figure 3. Colored net for the Ada program with safe allocation

One can remark that seats are not distinguishable and that a philosopher asks for **a** seat and not for a particular seat as he does for the chopsticks. This is modeled on the net by the color domain of place $Seat = \epsilon$ and by non valuated arcs around this place. Initially there is only $N-1$ free seats.

3.2. Simplifying this model

As previously, we apply some transitions agglomeration on this net : an agglomeration of transitions $give\,backRight$ and $give\,backSeat$, then an agglomeration of transitions $give\,backLeft$ and $give\,backRight.give\,backSeat$. We obtain the following net that is equivalent to the original for checking the liveness.

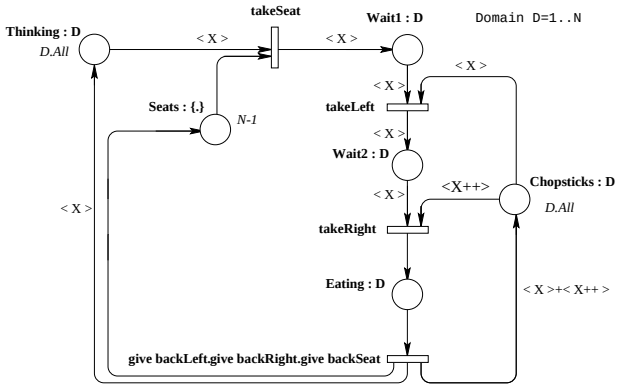

Figure 4. Reduced colored net for the safe allocation

3.3. Explaining the notion of controlled siphon

A siphon is a subset of places characterized by a particular structure : each transition that produces tokens in a siphon must also consume tokens from the siphon. Theoretically, a subset of places S is a siphon if and only if ${}^\bullet S \subseteq S^\bullet$. A siphon is minimal when it contains no other siphon as proper subset.

Siphons are also called "structural deadlock" because deadlock freeness of the net depends strongly on siphons marking : a siphon that becomes empty of tokens will never get again tokens since the transitions that produce token also consume tokens in the siphon. So, if this happen, all transitions of $S^\bullet$ will never be fireable again : the net is not live.

A siphon is said to be controlled [BPP96b] when it can never become empty of tokens, It has be proven in [BPP96b]that when all minimal siphons are controlled, the net is deadlock free : this is a necessary condition for liveness. Furthermore, for some sub classes of Petri nets (asymmetric choice or free choice nets, ...), it is

proved that this is also a sufficient condition for liveness : if all minimal siphons are controlled then the net is live [BPP96b].

In the net of figure 4 there are the minimal siphons :

- $S1_x = \{Thinking(x), Wait1(x), Wait2(x), Eating(x)\}$
- $S2 = \{seat\} \cup_{x \in D} \{Wait1(x), Wait2(x), Eating(x)\}$
- $S3 = \cup_{x \in D} \{Chopsticks(x), Eating(x)\}$

3.4. Defining places invariants

A place invariant defines a subset of places such that the global number of tokens in these places remains constant. Theoretically, a mapping $f \in P^{\mathbb{Z}}$ is a place invariant if and only if $\forall M \in Acc(N, M_0), \sum_{p \in P} f(p).M(p) = \sum_{p \in P} f(p).M_0(p) = cst$. Such a mapping is generally represented by a formal sum of places : $f \equiv \sum_{p \in P} f(p).p$.

In this net (fig. 4), we can compute the three sets of places invariants :

- $\forall x \in D, F1_x \equiv Thinking(x) + Wait1(x) + Wait2(x) + Eating(x) = cst1_x(M_0) = 1$
- $\forall x \in D, F2_x \equiv Chopsticks(x) + Wait2(x) + Eating(x) + Eating(x--)) = cst2_x(M_0) = 1$
- $F3 \equiv Seat + \sum_{x \in D}[Wait1(x) + Wait2(x) + Eating(x)] = cst3(M_0) = N - 1$

The first family of invariant $F1_x$ expresses that a philosopher x is either thinking, or waiting for the first chopstick, or waiting for the second chopstick or eats. These invariants characterize the process structure of philosophers.

The second family of invariants, $F2_x$, is related to the use of chopsticks : a chopstick x is either free, or is taken by the philosopher x which is waiting for its second chopstick, or is taken by the philosopher x or $x - 1$ which is eating. These invariants prove that a chopstick can not be taken by two philosophers.

The last invariant counts the number of free seats : a seat is either free or used by a philosopher which is waiting for the first or the second chopstick or which is eating. This invariant proves that there is at least one philosopher that is thinking.

3.5. Proving the liveness of the net

This net is an asymmetric choice net [BPP96b] since, if we consider every pair of places (p, q) then $p^\bullet \cap q^\bullet \neq \emptyset \Longrightarrow p^\bullet \subseteq q^\bullet$ or $q^\bullet \subseteq p^\bullet$. For this class of net a necessary and sufficient condition of liveness is that the minimal siphons are controlled, i.e. they never become empty of tokens.

Invariant $F1_x$ ensures that siphon $S1_x$ can not become empty of tokens since there is either a token in place $Thinking(x)$ or in place $Wait1(x)$ or in place $Wait2(x)$ or in place $Eating(x)$. So for any value of x, $S1_x$ is a controlled siphon. In the same way, $S2$ is controlled by the places invariants $F3$.

The non trivial point is to prove that the siphon $S3$ is controlled.

If we compose the invariants $F2$ and $F3$ we obtain a new places invariant $F4 = \sum_{x \in D} F2_x - F3$; $F4 \equiv \sum_{x \in D} Chopsticks(x) + \sum_{x \in D} Eating(x) = 1 + Seat + \sum_{x \in D} Wait1(x)$

This invariant tells us that the number of free chopsticks plus the number of eating philosophers is always equal to the number of free seats plus the number of philosophers waiting for their own chopstick plus 1. So, there is always at least either a free chopstick or a philosopher that is eating. This implies that the deadlock condition can not occur : the siphon $S3$ cannot become empty of token. We can then conclude that the net is live.

3.6. About starvation

The model of the solution does not take into account the scheduling policy of Ada95 regarding protected objects access. In particular, Ada95 ensures that at the end of the execution of an entry body (or a procedure body) of a protected object all barriers which have queued tasks are reevaluated thus possibly permitting the processing of an entry call which had been queued on a false barrier.

As we did not model this behavior, we cannot make an automatic proof of the starvation prevention of the solution. We will see in the next section how such a proof can be done.

However, suppose that a philosopher x, can never reach the state eating. This implies that either it can not obtain a seat or it can not obtain one of the two chopsticks.

Using places invariants computed previously and liveness of the net, we can ensure that in the first case (x is waiting for a seat) all other philosophers are either eating or waiting for one chopstick. As soon as a philosopher $y \neq x$ will return in state thinking, then all the barriers of the protected object Chaise will be reevaluated and thus permitting the philosopher x to get a seat before y can try to get again a seat. So, a philosopher x can not wait infinitely for a seat.

In the same way, suppose that a philosopher x can never obtain the first chopstick and waits in place

$Wait1$. Place invariant $F2_x$ implies that the chopstick x is owned by the philosopher $x-1$ which is eating. As philosopher $x-1$ must first give back its two chopsticks before trying to take again the chopstick x, entry barriers are reevaluated when philosopher $x-1$ releases its chopsticks, and then philosopher x gets its first chopstick.

Suppose at last that x is waiting for its second chopstick (thus $Wait2(x) = 1$ and $Chopstick(x+1) = 0$) can never obtain it. Using again invariant $F2_x$ (instantiated for x and $x+1$), we obtain that philosopher $x+1$ is either waiting for chopstick $x+2$ (in state $Wait2$) or is eating. If $x+1$ is eating, using the same argumentation as previously, when it releases its resources, entry barriers are reevaluated and philosopher x obtain its second chopstick. If $x+1$ is waiting for chopstick $x+2$ in state $Wait2$, there two cases: either $x+1$ will get in a future this chopstick or it may never get it. In the first case, as soon as $x+1$ gets chopstick $x+2$ it goes in state *Eating* and we have already studied this case. In the other case, using same argumentation, chopstick $x+2$ is owned by philosopher $x+2$ that is infinitely waiting for chopstick $x+3$ which is owned by philosopher $x+3$ that is infinitely waiting for chopstick $x+4$ and so on. Nevertheless, such a case is not possible since siphon $S3$ is controlled (there is no deadlock in the net). So, necessarily, there exists a $k > 0$ such that chopstick $x+k$ is owned by philosopher $x+k$ which is eating and when it will release its chopsticks, philosopher $x+k-1$ will access state *Eating* and so on until x access state *Eating*.

So, the reevaluation of entry barriers used in Ada95 and the deadlock freeness of the net ensure that this solution preserves of the starvation.

4. A solution with deadlock and starvation prevention and with information hiding

One drawback of the second solution is that philosophers have to know the way resources are managed. Changing the resources allocation policy leads to rewrite the code of the philosopher task. Ada95 allows the construction of an unique allocator that combines advantageously the first and second approach.

This allocator would be called by the philosophers according the scheme of the first solution and would process requests using the algorithm proposed in the second solution.

In Ada 95 the requeue statement enables a request to be processed in two or more steps. 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.

Entries visible from the philosophers are `demander(X)` (– i.e. ask(X)) and `conclure(X)` (– i.e. conclure(X)). Internally, the allocator has a family of N private entries `P_ask(X)`. The allocator processes only $N-1$ concurrent requests from `demander` entry point. A request is processed in several steps. At the end of each step, the processing may be postponed by putting the request in an entry queue. When the allocator accepts a request `demander(x)`, it first tries to allocate the two chopsticks needed. In case of success, the processing ends here. Else, the allocator tries to give one of the chopsticks and put the philosopher in the entry queue of the other chopstick (requeued on entry `B`). When it cannot allocate anything, it puts the philosopher X in the entry queue of the chopstick X.

To implement the algorithm, one needs to: count the number of request in the module (variable seat), know and update the status of the chopsticks (array chopsticks), store each request state (array step) where state can take the following values: init, nochopsticks, onechopstick, twochopsticks.

```
procedure le_bon_repas is
        N : constant integer := 5;
        type id is mod N;
        type tab_b is array(id) of boolean;
        type INOT is
             (Init, NoChopstick,
                    OneChopstick, TwoChopsticks);
        type tab_INOT is array(id) of INOT;

    protected repas is
        entry demander( X : in id);
        procedure conclure(X : in id);
        private
            chaise : integer := N - 1;
            baguette : tab_b := (others => true);
            etape : tab_INOT := (others => Init);
            entry B(id)(X: in id);  -- entry family
    end repas;

    protected numero is
        procedure unique(resultat: out id);
        private compte : id := id'first;
    end numero;

    task type philo;
    philosophe: array (id) of philo;

    task body philo is ego : id;
    begin
        numero.unique(ego);
        loop
            repas.demander(ego);
            mange(ego);
            repas.conclure(ego);
        end loop;
    end philo;
    protected body repas is separate;
    protected body numero is separate;
begin null;
end le_bon_repas;

-----------------------------------------------------------------
separate(le_bon_repas)
```

```
protected body numero is
    procedure unique(resultat : out id) is
    begin
        resultat := compte;
        compte := compte+1; -- addition modulo N
    end unique;
end numero;

-------------------------------------------------------------
separate(le_bon_repas)
protected body repas is
    entry demander(X :in id) when chaise > 0 is
    begin
        chaise := chaise - 1;
        if baguette(X) and baguette(X+1) then
          baguette(X) := false; baguette(X+1) := false;
          etape(X) := TwoChopsticks;
        elsif baguette(X) then
          baguette(X):=false; etape(X):=OneChopstick;
          requeue B(X+1);
        elsif baguette(X+1) then
          baguette(X+1):=false; etape(X):=OneChopstick;
          requeue B(X);
        else
          etape(X) := NoChopstick;
          requeue B(X);
        end if;
    end demander;

    procedure conclure(X: in id) is
    begin
        chaise := chaise + 1;
        baguette(X) := true; baguette(X+1) := true;
        etape(X) := Init;
    end conclure;

    entry B(for i in id)(X: in id) when baguette(i) is
    begin
        baguette(i) := false;
        if etape(X) = NoChopstick then
            etape(X):=OneChopstick; requeue B(X+1);
        else
            etape(X) := TwoChopsticks;
        end if;
    end B;
end repas;
```

Ada program 3 : program for safe allocation and information hiding

The following colored net (fig. 5) models this solution.

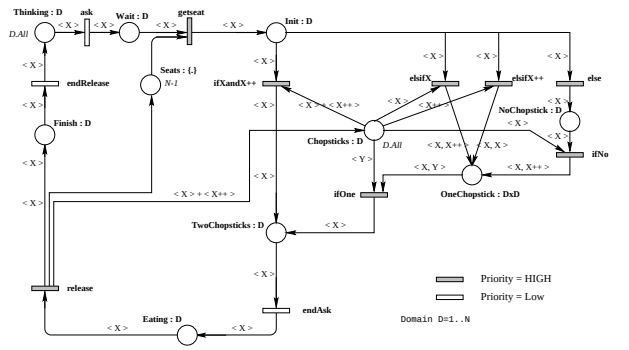

Figure 5. Colored net of Ada program 3

One can note that in this model a priority is associated to each transition (priority high or low). A transition of low priority can only be fired when no transition of high priority is fireable. We use this semantic in order to take into account the Ada95 processing policy of protected objects: at the end of a protected call, already queued entries (whose barriers are true) take precedence over new calls.

4.1. Proving deadlock prevention of this solution

Using same tools as previously (reductions theory, places invariants and controlled siphon property) we can prove that the model associated to this solution is a live net.

Applying reductions on this net, we obtain the following reduced net (fig. 6).

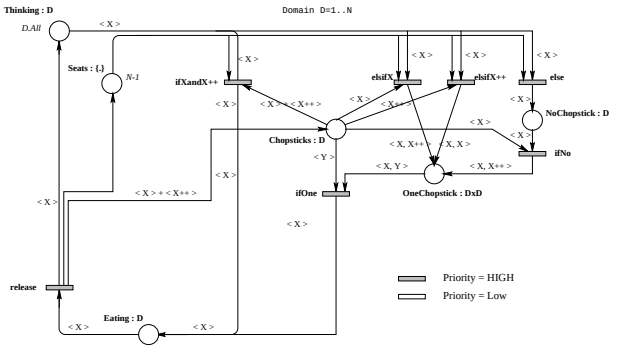

Figure 6. Reduced net of the Ada program 3

If we compute the places invariants and the siphons of the net, we obtain that only one siphon is not a place invariant (and may be not controlled): $S = \cup_{x \in D} \{Eating(x), Chopsticks(x)\}$ This siphon identifies a deadlock possibility: all philosophers take their first chopstick and then there is no available chopstick (no one can eat) and there is no philosopher that can give back a chopstick.

However, the places invariant F

$$\sum_{x \in D} Chopsticks(x) + \sum_{x \in D} Eating(x) = 1 + Seat + \sum_{x \in D} NoChopstick(x)$$

ensures that the siphon S contains at least one token. As the net is an asymmetric choice net (place *Seats* "controls" transitions $ifXandX++$, $elsifX$, $elsifX++$ and $else$) and as all siphons of the net are controlled we can conclude that the net is live.

4.2. Proving starvation prevention of this solution

In order to prove that the model does not enable infinite sequences for which a particular philosopher can not success to eat, as the system is finite,

the easiest way is to use methods based on temporal model checking. Many algorithms have been developed depending on property representation [Wol89, Flo96]. For instance, checking the absence of starvation of the model is equivalent to check (for $x_0 \in D$) the CTL^* [EPS93] formula [3] $\forall G(M(Wait(x_0)) = 1 \Longrightarrow \forall FM(Eating(x_0)) = 1)$ which means that all path issued from a state at which the philosopher x_0 is waiting for a seat includes a state in which the philosopher x_0 is eating.

The major problem limiting the use of these methods is the state space explosion. Different approaches are generally used to combat this problem. One can cite the methods based on Binary Decision Diagram [PRCB96], unfolding [ERV96] or the symbolic reachability graph [CDFH90] that permit to define a concise graph representation.

5. Conclusion

Programming preference control in ADA 95 leads very naturally to implement the resource management as an automaton with state transitions. This description can be directly modeled in Petri nets or colored nets, allowing to prove the correctness of the implementation. This method has been shown for the paradigm of the dining philosophers and different examples of proofs with Petri nets have been given.

It can be generalized to any client-server relationship where a server receives requests of different size and from different clients, and where all service decisions are taken by the server. The decisions may rely on the history of individual clients, or on the global history of the group of clients, or on the history of resource allocation; they may rely on individual requests or on the concomitance of requests of several users (for example a communication channel may need to be requested by both end users; another example is resource allocation for a client presenting a request which has to be authentified by a third party); they may still rely on claims of the behavior of future resource requests (as in the banker's algorithm for deadlock avoidance). The server can control the resource allocation by postponing the client request until some state of its automaton is reached.

References

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

[BLW87] A. Burns, M. Lister, and A. Wellings. A review of Ada tasking. In *LNCS*. Springer-Verlag, 1987.

[BPP96a] K. Barkaoui and J.F. Pradat-Peyre. *Introduction aux Reseaux de Petri. Cours Programmation et validation des applications concurrentes.* Polycopie CNAM, 1996.

[BPP96b] 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.

[CDFH90] C. Chiloa, C. Dutheillet, G. Franceschinis, and S. Haddad. On well-formed colored nets and their symbolic reachability graph. In *proc. of the 11th International Conference on Application and Theory of Petri Nets*, Paris-France, June 1990.

[CHP91] J.M. Couvreur, S. Haddad, and J.F. Peyre. Computation of generative families of semi-flows in two types of colored net. In *proc of the 12th International Conference on Application and Theory of Petri-Nets*, Aarhus, Denmark, June 1991.

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

[Elr88] T. Elrad. Comprehensive scheduling control for Ada tasking. In *Proceedings of the Second International Workshop on Real-Time Ada Issues*, pages 12–19, 1988.

[EPS93] A.E. Emerson and A. Prasad Sistl. Symmetry and model checking. In *proc. of the 5th conference on Computer Aided Verification*, June 1993.

[ERV96] J. Esparza, S. Romer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proc. of the 2nd Int. Workshop TACA'96*, number 1055 in LNCS, pages 87–106. Springer-Verlag, 1996.

[3] CTL^* is a language of expression of temporal logic properties and can be viewed as an unification of the Linear Time Logic (LTL) and the Computational Tree Logic (CTL)

[Flo96] G. Florin. *Introduction à la logique temporelle. Cours Programmation et validation des applications concurrentes.* Polycopie CNAM, 1996.

[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.

[Int95a] Intermetrics Inc. *Ada 95 Rationale*, 1995.

[Int95b] Intermetrics Inc. *Ada 95 Reference Manual*, 1995.

[Kai96] C. Kaiser. *Cours Programmation et validation des applications concurrentes.* Polycopie CNAM, 1996.

[PRCB96] E. Pastor, O. Roig, J. Cortadella, and R.M. Badia. Petri nets analysis using boolean manipulation. In Valette R., editor, *Petri Nets, Theory and Application*, number 815 in LNCS, pages 416–453. Springer-Verlag, 1996.

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

[WKT84] A. Wellings, D. Keeffe, and G. Tomlinson. *A Problem with Ada and Resource Allocation.* ACM SIGAda Ada Letters III(4), pp. 112-123, 1984.

[Wol89] P.. Wolper. On the relation of programs and computation to models of temporal logic. In *Proc. of Temporal Logic in Specifications, 89*, number 398 in LNCS, pages 87–106. Springer-Verlag, 1989.