Corrigé du Travaux Dirigés n°2
Ingénierie des protocoles
LOTOS

Question 1. Modélisation d'une machine à café

Question 1.1. Modélisation d'une machine simple

La modélisation de la machine simple est donnée par le processus LOTOS suivant :

```
process machine1 [jeton, thé, café, gobelet, fin] :=
  jeton ;
  (
  thé ; i ; exit
  []
  café ; i ; exit
  ) >> gobelet ; fin ; exit
endproc
```

Cette spécification attend une action sur sa porte jeton (plus précisément, attend de pouvoir se synchroniser avec l'environnement sur cette porte), puis au choix attend une action sur la porte thé ou café, puis exécute une action interne i, puis exécute gobelet et fin.

L'action interne « `i` » exécutée par le processus de fabrication modélise l'ensemble des actions internes faites pour fournir la boisson. La modélisation de la machine à café proposée ici s'abstrait de ces actions. En d'autres termes, elles sont « cachées » par « `i` ».

***Remarque** : on aurait pu modéliser le processus machine1 par :*

```
process machine1-v2[jeton, thé, café, gobelet, fin] :=
      ((jeton ; thé ; i ; exit)
      []
      (jeton ; café ; i ; exit))
      >> gobelet ; fin ; exit
endproc
```

Il faut cependant remarquer que cette modélisation n'est pas équivalente à la première. Ici, le choix entre thé et café est fait de façon non-déterministe au moment où on entre un jeton. A l'inverse, dans la première modélisation, le choix entre thé et café est fait après l'entrée du jeton. Ainsi, l'utilisateur peut entrer son jeton et avoir encore le choix de sa boisson (ce qui est quand même mieux). La bonne modélisation est donc la première.

Nous noterons dans la suite

```
process fabrique_the_cafe [thé, café, gobelet, fin] :=
  (
  thé ; i ; exit
  []
  café ; i ; exit
  ) >> gobelet ; fin ; exit
endproc
```

Le processus de la machine1 devient alors :

```
process machine1 [jeton, thé, café, gobelet, fin] :=
  jeton ;
  fabrique_the_cafe [thé, café, gobelet, fin]
endproc
```

Question 1.2. Modélisation d'une machine plus réaliste

On souhaite modifier la machine pour qu'elle fonctionne en boucle. Après avoir terminé sa livraison de thé ou café, elle doit se remettre en attente d'une autre commande.

Une solution très simple consiste à exprimer une boucle sur la spécification de la machine1. Dès que celle-ci termine, elle recommence. Cela s'exprime par un processus récursif.

```
process machine2 [jeton, thé, café, gobelet, fin] :=
  machine1[jeton, thé, café, gobelet, fin]  >> machine2[...]
endproc
```

Dès que la machine1 est terminée (après l'action fin), elle reboucle sur elle-même.

***Remarque** : dans un processus récursif* **process** `toto[a,b] :=` … `toto[...]` … **endproc**, *dans l'instanciation* `toto[...]`, *les « … » signifie que le processus toto est rappelé avec les même portes (ici, a et b).*

Question 1.3. Modélisation d'une machine encore plus réaliste

On souhaite remplacer le jeton par de l'argent. On peut proposer deux modèles. Le premier abstrait en FULL LOTOS, le second plus concret en BASIC LOTOS.

1.3.1. Modèle abstrait en FULL LOTOS

Dans ce modèle on suppose que l'usager introduit une somme d'argent par une porte « pin », et récupère sa monnaie par une porte « pout ». L'idée est alors de rajouter un processus « `monnaie-full-lotos` » qui collectera la somme de 150 centimes d'euros, puis qui rendra éventuellement la monnaie, et qui produira un jeton virtuel vers le module « `machine1` ». De ce fait, ce dernier module reste inchangé.

Le nouveau module « `monnaie-full-lotos` » doit présenter une porte d'entrée « pin » et une porte de sortie « pout ». Tant que la somme reçue sur « pin » est insuffisante, le processus reboucle pour attendre l'arrivée d'autre argent. En revanche, si la somme donnée est supérieure à 150 centimes d'euros, il rend la différence sur « pout » et produit un jeton. La spécification de ce processus est :

```
process monnaie-full-lotos[pin, pout, jeton](V : nat)  :=
      pin ?x : nat ;
      [x+V < 150] -> monnaie-full-lotos[...](x+V) ; exit
      []
      [x+V = 150] -> jeton; pout !0 ; exit
      []
      [x+V > 150] -> jeton; pout !(x+V - 150) ; exit
endproc
```

Le processus complet est alors l'assemblage de « `monnaie-full-lotos` », et « `machine1` ». Ce processus complet est décrit par :

```
process machine3-full-lotos[pin, pout, thé, café, gobelet, fin] :=
hide jeton in
      (
      monnaie-full-lotos[pin, pout, jeton](0)
      |[jeton]|
      machine1[jeton, thé, café, gobelet, fin]
      )
      >> machine3-full-lotos[...]

where …
```

1.3.2. Modèle concret en BASIC LOTOS

Dans ce modèle, on s'intéresse aux pièces reçues et rendues. Les portes du système sont donc ces pièces. Le prix de la boisson est 150 centimes d'euros et les pièces acceptées sont 200 centimes d'euros, 100 centimes d'euros et 50 centimes d'euros. La machine doit donc rendre la monnaie. Le nouveau module « `monnaie-basic-lotos` » doit présenter autant de portes que de types de pièces acceptées en entrée, autant de portes que de types de pièces rendues en sortie, et une porte jeton (pour synchronisation avec le processus « `interface` »). La spécification de ce processus est :

```
process monnaie-basic-lotos[in_2, in_1, in_50c, out_1, out_50c, jeton]  :=
      (in_2 ; out_50c ; exit)
      []
      (in_1 ; (in_2 ; out_50c ; out_1 ; exit)
        []
        (in_1 ; out_50c ; exit)
        []
        (in_50c ; exit)
      )
      []
      (in_50c ; (in_2 ; out_1 ; exit)
           []
           (in_1 ; exit)
          []
           (in_50c ; (in_2 ; out_1 ; out_50c ; exit)
                     []
                     (in_1 ; out_50c ; exit)
                     []
                     (in_50c ; exit)
          )
      )
      >> jeton ; exit
endproc
```

Le processus complet est alors l'assemblage de « `monnaie-basic-lotos` », et « `machine1` ». Ce processus complet est décrit par :

```
process machine3-basic-lotos[in_1, in_2, in50c, out_1, out_50c,
                          thé, café, gobelet, fin] :=
hide jeton, ordre_thé, ordre_café in
      (
      monnaie-basic-lotos[in_2, in_1, in_50c, out_1, out_50c, jeton]
      |[jeton]|
      machine1[jeton, thé, café, gobelet, fin]
      )
      >> machine3-basic-lotos[...]

where …
```

***Remarque** : dans les deux cas, le processus « `monnaie-xxx-lotos` » termine après avoir produit le jeton. Il n'est donc pas possible d'introduire de nouvelles pièces sans que la fabrication soit terminée et que la boucle globale ait fonctionné.*

Question 1.4. Modélisation d'une machine définitivement plus réaliste (en Full LOTOS)

On demande maintenant de tenir compte du nombre limité de thé et de café disponible et de faire en sorte que le choix de la boisson manquante soit impossible. Pour ce faire, on reconsidère la machine de sorte que l'utilisateur externe ne puisse pas sélectionner un thé, si le processus « `machine` » n'est pas prêt à fabriquer un thé.

Pour modéliser le fait que le thé est épuisé, on fait en sorte la machine ne puisse pas offrir l'action thé à l'environnement externe (et donc pas de synchronisation possible sur cette porte). Pour ce faire, on introduit dans le processus « `machine` » une branche correspondant à cet état (pas de thé), et prise de façon non-déterministe sur une action interne (l'action interne « `i` » cache ici le comptage ou la gestion du stock ; il s'agit en d'autres termes d'une abstraction non déterministe d'opérations réelles déterministes). En généralisant à l'action café, on obtient le processus :

```
process machin-elem4 [thé, café, gobelet, fin] :=
        jeton ;
        (
          fabrique_the_cafe[thé, café, gobelet, fin]
        []
          (i ; (thé ; gobelet ; i ; fin ; exit))
        []
          (i ; (café ; gobelet ; i ; fin ; exit))
        []
          (i ; exit)
      )
endproc
```

Ce processus modélise quatre cas possibles. Soit les deux actions thé et café sont possibles (le cas normal déjà modélisé à la question 1.1), soit après une action interne « i » qui modélise le fait qu'une boisson est testée épuisée, seuls le thé ou le café restent disponibles, soit les deux boissons sont indisponibles. L'action correspondant à la boisson épuisée étant impossible dans le processus ci-dessus, et comme cette action doit être synchronisée avec les actions de l'usager, elle est donc impossible à l'usager.

Les boissons pouvant être épuisées, il peut alors être judicieux (si on ne veut pas de violence inutile contre la machine), que l'utilisateur puisse récupérer son argent. Il est alors nécessaire d'introduire une porte « `remboursement` » comme un troisième choix après « `thé` » ou « `café` ». L'action remboursement conduit cependant à une action préemptive sur les processus en cours, notamment le processus de fabrication, et le remboursement de 150 centimes d'euros, puis retour dans l'état initial de la machine. On modélise une telle action préemptive par un processus de préemption placé à l'extérieur de la boucle nominale :

```
process machine4 [pin, pout, thé, café, remboursement, gobelet, fin] :=
hide jeton in
    ((
    monnaie-full-lotos[pin, pout, jeton](0)
    |[jeton]|
    machine-elem4[jeton, thé, café, gobelet, fin]
    )
    >> machine4[...])
    [> remboursement ; pout !150 ; machine4[...]
endproc
```

Ce processus contient une boucle nominale active lorsque les deux processus monnaie-full-lotos, et machine-elem4 se termine normalement (par un exit), et une boucle externe préemptive lorsque l'usager appuie sur remboursement. Il est nécessaire de passer par une telle boucle préemptive supérieure car le processus machine-elem4 n'attend pas remboursement et donc ne termine pas sur arrivée de cette action. Il faut forcer sa terminaison par l'opérateur de préemption.

Cette machine présente cependant un défaut (grave pour le vendeur de boissons) : l'action remboursement est possible à tout moment, y compris avant d'avoir introduit de l'argent et même après avoir obtenu une boisson. Il est donc nécessaire de modifier la machine.

Une première modification porte sur le monnayeur qui devra accepter le remboursement et rendre l'argent déjà introduit si cette action est sélectionnée :

```
process monnaie-full-lotos5[pin, pout, jeton, remboursement](V : nat)  :=
        (remboursement ; pout !V ; monnaie-full-lotos4[...](0))
      []
        pin ?x : nat ;
            ([x+V < 150] -> monnaie-full-lotos5[...](x+V) ; exit
            []
            [x+V = 150] -> jeton; pout !0 ; exit
            []
            [x+V > 150] -> jeton; pout !(x+V - 150) ; exit)
endproc
```

Ce processus (full LOTOS) est identique au premier processus « monnaie-full-lotos » à ceci près qu'il contient une branche de choix supplémentaire sélectionnée sur remboursement et provoquant l'émission sur pout de la valeur V. Ensuite le processus se remet en attente d'une nouvelle action (une valeur sur pin ou un remboursement).

La deuxième modification porte sur le corps de fabrication qui doit lui aussi accepter l'action remboursement :

```
process machin-elem5 [thé, café, gobelet, fin, remboursement, pout] :=
        jeton ;
      (
            (
            (fabrique_the_cafe[thé, café, gobelet, fin]
            []
            (remboursement ; exit))
          []
            (i ; (thé ; gobelet ; i ; fin ; exit)
                 []
                 (remboursement ; exit)
             )
          []
            (i ; (café ; gobelet ; i ; fin ; exit)
                 []
                 (remboursement ; exit)
             )
         )
       |[gobelet, remboursement]|
         (
             (gobelet ; exit)
           []
             (remboursement ; pout !150 ; exit)
         )
      )
endproc
```

En plus de remboursement, ce nouveau processus doit également avoir accès à la porte pout de manière à y envoyer si nécessaire la somme de 150 cts d'euro. Ce processus est composé de deux sous-processus en parallèle. Le premier correspondant en première approximation à la machine-elem4, c'est-à-dire la machine pouvant le plus avoir de thé ou de café, dans laquelle a été ajouté la possibilité de demander un remboursement, et d'un second processus pouvant soit faire gobelet, soit une action de remboursement. Si l'usager externe appuie sur remboursement, les deux processus étant synchronisés sur cette action, ils l'effectue en même temps. le premier sous-processus termine aussitôt tandis que le deuxième sous-processus rend la monnaie sur pout puis termine. Si l'usager sélectionne une boisson disponible, alors le premier sous-processus émet gobelet vers l'environnement externe mais aussi vers le second sous-processus qui termine alors instantanément.

Remarquons que par rapport aux versions précédentes, cette machine accepte le remboursement à tout moment, y compris s'il y a encore une boisson disponible.

La nouvelle machine devient alors :

```
process machine5 [pin, pout, thé, café, remboursement, gobelet, fin] :=
hide jeton in
    ((
    monnaie-full-lotos5[pin, pout, jeton, remboursement](0)
    |[jeton]|
    machine-elem5[jeton, thé, café, gobelet, fin, remboursement, pout]
    )
    >> machine5[...])
endproc
```

Question 2. Protocole Producteur – Consommateur

Question 2.1. Cas du Médium = buffer à une case

2.1.1. Le système global

Le système est composé de trois processus : un producteur, un consommateur, et un médium. Le producteur attend un ordre de démarrage, puis émet ses N données vers le médium, puis termine en produisant vers la couche supérieure un signal off-prod. De même, le consommateur attend un ordre de démarrage puis se met en réception sur le médium. Après avoir consommé l'ensemble des données du producteur, le consommateur se déconnecte et émet le signal off-cons.

Ces trois processus communiquent par échange de signaux sur des portes internes : in-data pour l'émission des données par le producteur vers le médium, out-data pour l'envoi de ces données du médium vers le consommateur.

De plus, le producteur après avoir émis ses N données doit signaler la fin de l'émission au consommateur en produisant un signal de déconnexion. Ce signal transitera par la porte in-deconnect du producteur vers le médium et par la porte out-deconnect du médium vers le consommateur.

Nous choisissons ici de décrire une version du producteur paramétré par le nombre N de données à émettre. Ce nombre N est un paramètre global du système.

```
process system [on-prod, on-cons, off-prod, off-cons](N : nat) :=
hide in-data, out-data, in-deconnect, out-deconnect in

    producteur[on-prod, in-data, in-deconnect, off-prod](N)
    |[in-data, in-deconnect]|
    medium[in-data, out-data, in-deconnect, out-deconnect]
    |[out-data, out-deconnect]|
    recepteur[on-cons, out-data, out-deconnect, off-cons]

endproc
```

2.1.2. Le producteur

Le processus producteur est tout d'abord en attente sur la porte on puis boucle de façon récursive jusqu'à émission des N data, puis émet le signal de deconnexion, et signale enfin sa terminaison par émission du signal off vers la couche supérieure. Ce processus peut être décrit par une séquence :

```
process producteur [on, data, deconnect, off](N : nat) :=
    on ; prod[data](N); deconnect; off; exit
endproc
```

où prod[data](N) est un processus récursif terminant lorsque N=0 :

```
process prod [data](N : nat) :=
    [N>0] -> data ; prod[data](N-1)
    []
    [N=0] -> exit
endproc
```

2.1.3. Le consommateur

La spécification du consommateur dépend des hypothèses faite sur le médium. On supposera que le médium est fiable et qu'il respecte l'ordre des messages. En particulier, le message de déconnexion ne

doublera pas un message de donnée. Lorsque le consommateur recevra un message de déconnexion, alors il sera certain qu'il n'y a plus de données à consommer et pourra se déconnecter.

Sous cette hypothèse, le processus consommateur est tout d'abord en attente sur la porte on puis boucle de façon récursive en réception des data sur le médium (Note : le consommateur ne connaît pas le nombre N d'émission). Dès réception du signal de déconnexion, le consommateur termine en produisant le signal off vers la couche supérieure. Ce processus peut être décrit par une séquence :

```
process consommateur [on, data, deconnect, off] :=
    on ; cons[data][> deconnect; off; exit
endproc
```

La réception du signal deconnect provoque l'interruption définitive du processus cons. Le processus consommateur sera donc ainsi en attente sur data (à l'intérieur de cons) et sur deconnect.

Le processus cons s'écrit simplement comme une attente récursive infinie sur data :

```
process cons [data] :=
    data ; cons[data]
endproc
```

2.1.3. Le médium à une case

Reste alors à modéliser le médium en respectant l'hypothèse faite ci-dessus : médium fiable conservant l'ordre des messages.

Le médium est composé d'une seule ligne de transmission qui peut transmettre soit des données, soit le message de contrôle deconnect. Ceci peut s'exprimer par :

```
process médium [in-data, out-data, in-deconnect, out-deconnect] :=
    in-data ; out-data ; medium1[...]
    []
    in-deconnect ; out-deconnect ; medium1[...]
endproc
```

Ce processus est en attente soit sur in-data, soi sur in-deconnect, puis sur l'arrivée de l'un de ces deux messages produit le message de sortie correspondant, puis recommence.

Question 2.2. Cas du Médium = buffer à K case

Dans le cas d'un médium pouvant faire transiter jusqu'à K messages simultanément, on peut modéliser un tel médium comme la mise en parallèle de K médium simple à une case. Toutefois, une telle conception ne respecte plus l'hypothèse de conservation de l'ordre des messages. Il devient donc nécessaire de modifier le consommateur, et notamment de permettre à celui-ci de tester la présence ou l'absence de message dans le médium. On introduit donc une porte vide sur laquelle se synchroniseront le médium et le consommateur si le médium est effectivement vide.

Le medium s'exprime donc par :

```
process médium [in-data, out-data, in-deconnect, out-deconnect] :=
    ligne[in-data, out-data, in-deconnect, out-deconnect, vide]
    |[vide]|
    ligne[in-data, out-data, in-deconnect, out-deconnect, vide]
    |[vide]|
    …
    |[vide]|
    ligne[in-data, out-data, in-deconnect, out-deconnect, vide]
endproc
```

(avec K ligne en parallèle) où le processus ligne est défini par

```
process ligne [in1, out1, in2, out2, vide] :=
    in1 ; out1 ; ligne[...]
    []
    in2 ; out2 ; ligne[...]
    []
    vide ; ligne[...]
endproc
```

Sous l'hypothèse de ce nouveau médium, le consommateur doit tester si le médium est vide avant de se déconnecter. L'arrivée d'un message deconnect provoque alors le basculement vers un nouveau mode dans lequel le processus continue à consommer les données jusqu'à ce qu'il puisse faire l'action vide avec le médium.

```
process consommateur [on, data, deconnect, off, vide] :=
    on ; cons[data][>
    deconnect; cons[data] [> vide ; off; exit
endproc
```

La réception du signal deconnect provoque l'interruption du processus cons[data], puis à nouveau l'exécution cons[data] (lecture en boucle des data) jusqu'à ce que vide soit possible. L'action vide provoque alors la fin du processus cons[data], et par suite l'exécution de off puis la terminaison du processus consommateur.