4I507 - Master 1 Informatique 2015-2016
Spécialité STL
Programmation Concurrente, Réactive et Répartie

Quelques exemples d'Esterel

Index

0. Préliminaires

- Les exemples ci-dessous sont inspirés de
  - Description de comportements d'agents autonomes évoluant dans des mondes virtuels de Nadine Richard (Thèse présentée pour obtenir le grade de docteur de l'École Nationale Supérieure des Télécommunications). Chapitre 4 (page 63-87).
  - The Esterel v5 Language Primer – Version v5_91 de Gérard Berry.

- Pour compiler avec la version Esterel v5_92 installée dans /Vrac/PC2R/esterelv5_92 (penser à changer le chemin si ailleurs) :
  - ```
    export ESTEREL=/Vrac/PC2R/esterelv5_92
    export PATH=$ESTEREL/bin:$PATH:.
    esterel nomA.strl -B nomB -simul
    ```
    ```

```
gcc -m32 -o nomC nomB.c \
        -I $ESTEREL/include -L $ESTEREL/lib -lcsimul
```

- Pour la simulation, l'utilisateur utilise un « ; » pour représenter l'instant d'un `tick`. Il peut insérer un ou plusieurs signaux d'entrée devant (à gauche d') un « ; » pour simuler la présence de ces signaux du `tick` correspondant et il termine la simulation par un « . ».

- Toutes remarques sont les bienvenues et doivent être envoyées à LIEU Choun Tong (choun-tong.lieu@upmc.fr).

1. module

module *nom* **:**
Déclarations d'interface
Instructions
end module

Exemple1.strl :

```
module Exemple1 :

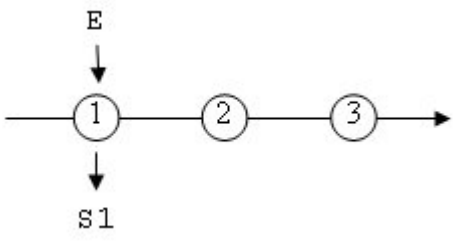
  input  E;                 % déclarations d'interface :
  output S1, S2;            %   signal d'entrée E
                            %   signaux de sortie S1 et S2

  present E then            % si E est dans le premier tick
      emit S1               %    ça émet S1
    else emit S2            %    sinon ça émet S2
  end present

end module
```

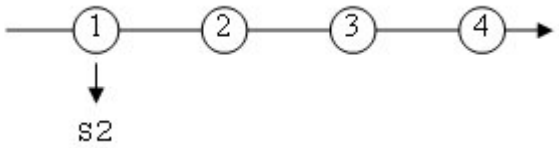
1er test :

```
Exemple1> E;;;
--- Output: S1
--- Output:
--- Output:
Exemple1>.
```



2ème test :

```
Exemple1> ;;;;
--- Output: S2
--- Output:
--- Output:
--- Output:
Exemple1>.
```



- Plusieurs modules peuvent être définis dans un seul fichier ou dans des fichiers séparés.
- Dans un module, on peut appeler un autre module par l'instruction **run** *nom_du_module* ou **run** *nom_du_module* **[** *correspndance_entre_interfaces_appelé_et_appelant* **]**.
- Tout appel croisé est interdit.
- Esterel choisit d'exécuter le premier module rencontré qui n'est pas appelé par les autres.
- Toutes les déclarations d'interface du module appelé doivent être couvertes par les déclarations d'interface ou les déclarations locales du module appelant.
  Virtuellement, on peut considérer comme si on remplace l'instruction **run** *xxx* par le corps du module *xxx*.
- Si les noms dans les déclarations de l'interface du module appelé se retrouvent déclarés (même nom et type) dans le module appelant, on peut utiliser la forme **run** *nom_du_module*.
- Sinon, on utilisera la forme **run** *nom_du_module* **[** *correspndance_entre_interfaces_appelé_et_appelant* **]** permettant de déclarer la correspondance entre les noms du module appelé et ceux du module appelant. On peut ne déclarer que les noms différents.

Exemple1_1.strl : plusieurs modules définis dans le même fichier

```
module Exemple1_1 :

 input  E;
 output S, S1 : integer, S2 : integer;
```

```
  [ present E then
       emit S1(1)
    else emit S2(2)
  end present
  ||
  run Exemple1_2
  ];

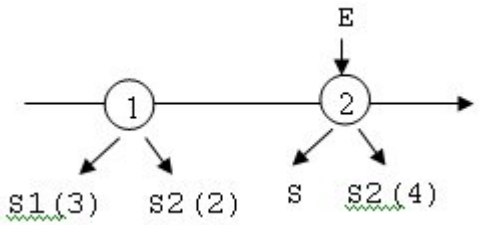
  emit S
end module

module Exemple1_2 :
  input  E;          % memes noms
  output S1 : integer, S2 : integer;

  emit S1(3);
  await E;
 emit S2(4);
end module
```

test :

```
Exemple1_1> ;E;.
--- Output: S1(3) S2(2)
--- Output: S S2(4)
```



test :

```
Exemple1_1> E;;E;.
--- Output: S1(3)
--- Output:
--- Output: S S2(4)
```

Exemple1_2.strl : avec quelques noms différents

```
module Exemple1_2_2 :
  input  E;          % noms différents
  output S2_1 : integer, S2_2 : integer;

  emit S2_1(3);
```

```
  await E;
  emit S2_2(4);
end module

module Exemple1_2_1 :
  input  E;
  output S, S1 : integer, S2 : integer;

  [ present E then
       emit S1(1)
    else emit S2(2)
  end present
  ||
  run Exemple1_2_2 [ signal S1 / S2_1,
                             S2 / S2_2 ]
  ];

  emit S
end module
```

Exemple1_3_1.strl : en deux fichiers séparés

```
% associe au fichier Exemple1_3_2.strl
% a compiler avec
% $ESTEREL/bin/esterel -simul \
%     Exemple1_3_1.strl \
%     Exemple1_3_2.strl -B Exemple1_3_12

module Exemple1_3_1 :
  input  E;
  output S, S1 : integer, S2 : integer;

  [ present E then
       emit S1(1)
    else emit S2(2)
  end present
  ||
  run Exemple1_3_2 [ signal S1 / S2_1,
                             S2 / S2_2 ]
  ];

  emit S
end module
```

Exemple1_3_2.strl :

```
% associe au fichier Exemple1_3_1.strl
% a compiler avec
% $ESTEREL/bin/esterel -simul \
%     Exemple1_3_1.strl \
%     Exemple1_3_2.strl -B Exemple1_3_12

module Exemple1_3_2 :
  input  E;
  output S2_1 : integer, S2_2 : integer;
```

```
  emit S2_1(3);
  await E;
  emit S2_2(4);
end module
```

2. input, output, inputoutput et signal : valués ou non.

- Tous les signaux peuvent être émis ou attendus.
- Les **input** sont des signaux que l'on peut saisir en entrée, que l'on peut émettre et dont l'émission n'est pas envoyée à la sortie.
- Les **output** sont des signaux que l'on peut émettre, que l'on ne peut pas saisir en entrée et dont l'émission est aussi envoyée en sortie.
- Les **inputoutput** sont des signaux qui se comportent comme des **input** et des **output**.
- Les **signal** ne sont ni des **input**, ni des **output** : on ne peut les saisir à partir de l'entrée et leur émission n'est pas envoyée à la sortie. **signal** permet de définir des signaux locaux à un bloc d'instruction :

```
signal les_signaux in
  corps
end signal
```

Exemple2_1.strl :

```
module Exemple2 :

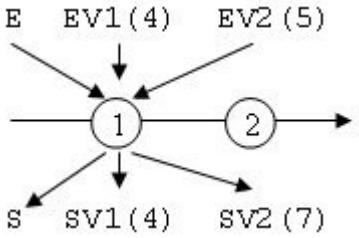
  input  E, EV1 : integer, EV2 := 3 : integer;
  output S, SV1 : integer, SV2 := 7 : integer;

  present E then
      emit S
  end present;
  present EV1 then
      emit SV1(?EV1)     % émet SV1 avec la valeur
  end present;      %    associée à EV1
  present EV2 then
      emit SV2(?EV2 + 1);
      emit SV2(?EV2 + 2) % seul ce dernier SV2 est émis
                         % sauf si on utilise
combine
  end present

end module
```

test :

```
Exemple2> E EV1(4) EV2(5);;.
--- Output: S SV1(4) SV2(7)
--- Output:
```



Exemple2_2.strl :

```
module Exemple2_2 :

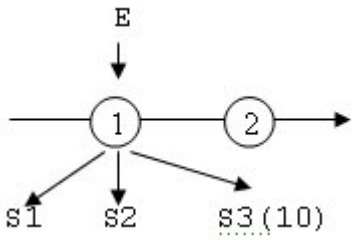
  input  E;
  output S1, S2, S3 : integer;

  signal LV1 : integer, L2 in % signaux locaux
    present E then
        emit S1;
        emit LV1(10)     % ça ne s'affiche pas en output
    end present;
    present S1 then % celui qui est émis ci-dessus
        emit S2
    end present;
    present LV1 then
        emit L2;
        emit S3(?LV1)
    end present
  end signal

end module
```

1er test :

```
Exemple2_2> E;;.
--- Output: S1 S2 S3(10)
--- Output:
```



2ème test :

```
Exemple2_2> LV1(10);;.
*** Error: not an input : LV1
--- Output:
```

3ème test :

```
Exemple2_2> S1;;.
*** Error: not an input : S1
--- Output:
```

3. **present** : instruction atomique non bloquant pour détecter dans le `tick` courant (c'est-à-dire dans le `tick` où l'instruction commence son exécution) la présence d'un signal singulier ou composé. C'est donc une instruction instantanée, sans attendre.

present *Expression_des_signaux*
then *Instructions*
else *Instructions*
end present

present *Expression_des_signaux*
then *Instructions*
end present

present *Expression_des_signaux*
else *Instructions*
end present

Exemple3_1.strl :

```
module Exemple3_1 :

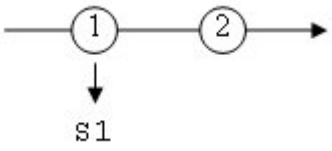
  input  E1, E2;
  output S1, S2, S3;
```

```
  present tick then % on est dans un tick
    emit S1;              % émet S1 dans ce tick
  end present;       %   sauf s'il y en a un autre S1
  present tick then % on est toujours dans ce même
    emit S1;              %   tick et il y en a un autre
S1
  end present;       %   c'est ce dernier qui est émis
  present E1 or E2 then
    emit S2;              % émet S2 dans ce même tick
  end present;
  present E1 and E2 then
    emit S3;              % émet S3 dans ce même tick
  end present;

end module
```

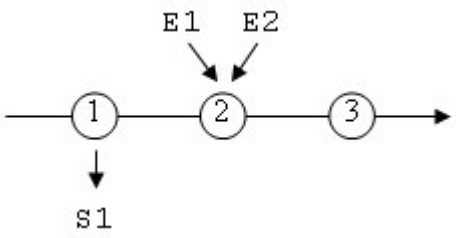
1er test :

```
Exemple3_1> ;;.
--- Output: S1
--- Output:
```



2ème test :

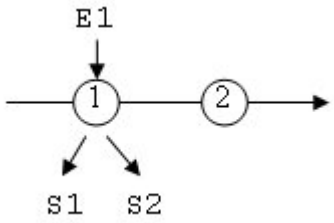
```
Exemple3_1> ; E1 E2;;.
--- Output: S1
--- Output:
--- Output:
```



3ème test :

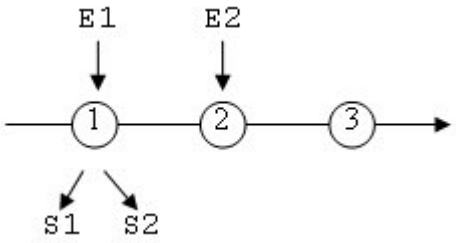
```
Exemple3_1> E1;;.
--- Output: S1 S2
```

```
--- Output:
```



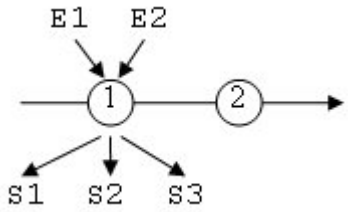
4ème test :

```
Exemple3_1> E1; E2;;.
--- Output: S1 S2
--- Output:
--- Output:
```



5ème test :

```
Exemple3_1> E1 E2;;.
--- Output: S1 S2 S3
--- Output:
```



Exemple3_2.strl :

```
module Exemple3_2 :

  input  E1, E2;
  output S1, S2, Stick;

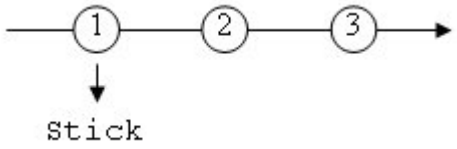
  present tick then
    emit Stick;
```

```
  end present;
  present E1 then
    emit S1
  end present;
  present tick then
    emit Stick;
  end present;
  present E2 then
    emit S2
  end present;
  present tick then
    emit Stick;
  end present;

end module
```

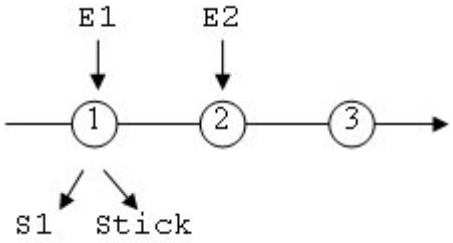
1er test :

```
Exemple3_2> ;;;.
--- Output: Stick
--- Output:
--- Output:
```



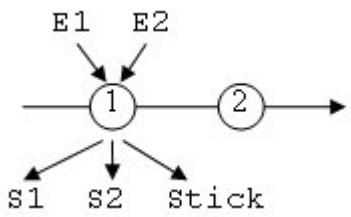
2ème test :

```
Exemple3_2> E1 ; E2 ;;.
--- Output: S1 Stick
--- Output:
--- Output:
```



3ème test :

```
Exemple3_2> E1 E2;;.
--- Output: S1 S2 Stick
--- Output:
```



4. **await** : instruction retardée (les signaux de l'instant courant ne sont pas pris en compte) et non-instantanée permettant d'attendre (dans les prochains `tick`) l'arrivée d'un signal en gelant (bloquant) l'exécution de son thread jusqu'à l'apparition du signal dans les prochains `tick`.

await *DelayExpression*

await *DelayExpression*
do *Instruction*
end await

await
case *DelayExpression*
...
case *DelayExpression* **do** *Instruction*
...
end await

avec

DelayExpression = *BracketedSignalExpression*

immediate *BracketedSignalExpression*

Expression BracketedSignalExpression

BracketedSignalExpression = *Expression_des_signaux*

[*Expression_des_signaux* **]**

avec

await immediate E;
≈ present E **else await** E **end present;**

Si E est présent dans le `tick` courant (c'est-à-dire dans le `tick` où l'instruction commence son exécution), l'instruction **await immediate** E n'est pas bloquant, sinon elle l'est.

Exemple4_1.strl :

```
module Exemple4_1 :

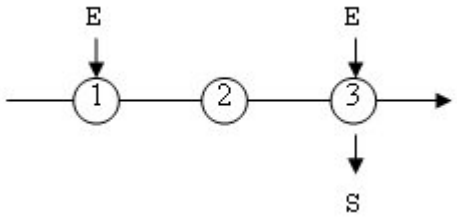
  input  E;
  output S;

  await E;
  emit S

end module
```

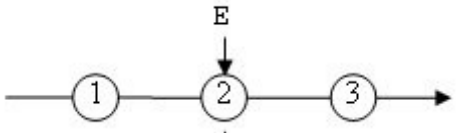
1er test :

```
Exemple4_1> E;;E;.
--- Output:
--- Output:
--- Output: S
```



2ème test :

```
Exemple4_1> ;E;;.
--- Output:
--- Output: S
--- Output:
```



Exemple4_2.strl :

```
module Exemple4_2 :

  input  E;
  output S;

  await (1 + 2) E;  % await E; await E; await E;
  emit S

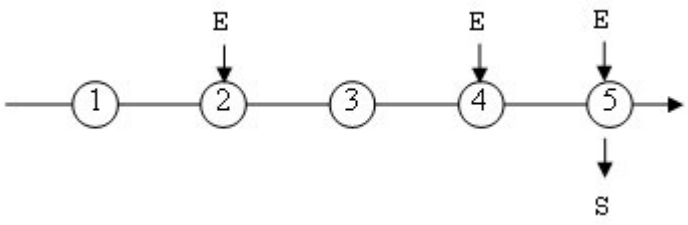
end module
```

1er test :

```
Exemple4_2> ; E E E ;.
--- Output:
*** Error: single signal input more than once: E
*** Error: single signal input more than once: E
```

2ème test :

```
Exemple4_2> ; E ;; E; E;.
--- Output:
--- Output:
--- Output:
--- Output:
--- Output: S
```



Exemple4_3.strl :

```
module Exemple4_3 :

  input  E1, E2, E3;
  output S1, S2, S3;

  await  E1;

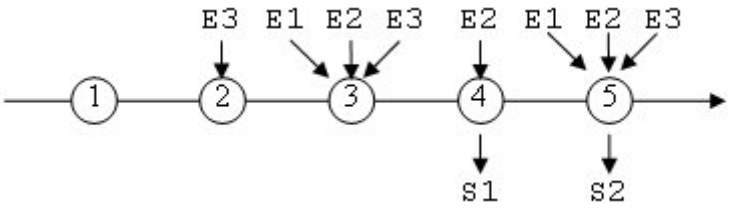
  await [not E1] do emit S1 end await;

  await
    case [E1 and E2] do emit S2
    case [E1 or E2] do emit S3
  end await

end module
```

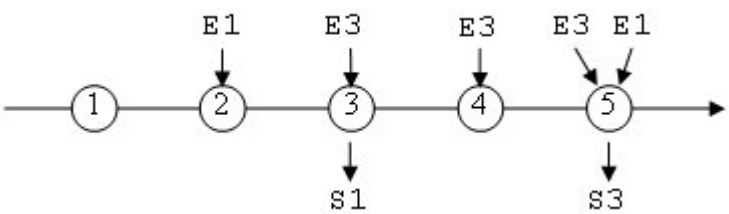
1er test :

```
Exemple4_3> ; E3; E1 E2 E3; E2; E1 E2 E3;.
--- Output:
--- Output:
--- Output:
--- Output: S1
--- Output: S2
```



2ème test :

```
Exemple4_3> ; E1; E3; E3; E3 E1;.
--- Output:
--- Output:
--- Output: S1
--- Output:
--- Output: S3
```



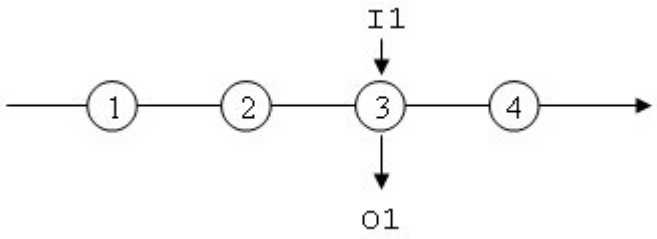
Exemple4_4.strl :

```
module Exemple4_4 :

  input  I1, I2;
  output O1, O2;
  signal S in
    await I1;
    emit I2;
    present I2 then emit O1 end present;
    emit S;
    await S do % await retardé, a laissé échapper le
signal S
      emit O2       %   courant pour attendre le prochain
S.
    end await       % C'est bloqué, car S ne peut être
donné en entrée
  end signal
end module
```

test :

```
Exemple4_4> ;; I1;;.
--- Output:
--- Output:
--- Output: O1
--- Output:
```



5. **Opérateur parallèle || pour la composition des instructions** : L'opérateur binaire de composition parallèle **||** est moins prioritaire que l'opérateur binaire de composition de séquence **;** . Ce qui explique l'utilisation des parenthèses **[]** pour regrouper les instructions. Ces deux opérateurs sont associatifs gauches. Ça fonctionne comme les opérateurs + et * pour les expressions arithmétiques.

Par exemple :

$inst_1$ **;** $inst_2$ **;** $inst_3$ **||** $inst_4$ **;** $inst_5$ est équivalent à **[[** $inst_1$ **;** $inst_2$ **] ;** $inst_3$ **] || [** $inst_4$ **;** $inst_5$ **]**

$inst_1$ **||** $inst_2$ **;** $inst_3$ **||** $inst_4$ **;** $inst_5$ est équivalent à **[** $inst_1$ **|| [** $inst_2$ **;** $inst_3$ **]] || [** $inst_4$ **;** $inst_5$ **]**

G || D est une instruction qui permet de lancer instantanément et simultanément deux threads en parallèle : le premier pour sa composante **G** et le deuxième pour sa composante **D**.

L'instruction **G || D** est terminée à l'instant où les composantes **G** et **D** sont terminées (au même moment ou l'une après l'autre).
Si l'une des deux composantes est terminée avant l'autre, **G || D** attendra la terminaison de l'autre pour terminer l'instruction.

Comme pour l'opérateur de composition de séquence **;**, on peut écrire $inst_1$ **||** $inst_2$ **||** ... **||** $inst_n$ (équivalent à **[...[**$inst_1$ **||** $inst_2$**] ||** ... **||** $inst_n$ **]**).

Exemple5_1.strl :

```
     module Exemple5_1 :

  input  E;
  output S : integer;

  await  E;

  var x := 10 : integer in
    [ x := x + 1; emit S(x); pause
      ||
      await E; x := x + 2; emit S(x)
    ];
    emit S(x)
  end var;

end module
```

Avec la commande esterel :

```
     $ /usr/local/esterelv5_92/bin/esterel -simul
    Exemple5_1.strl -B Exemple5_1
    *** strlic: Error #2.4.17: "Exemple5_1.strl", line 11,
    col 21
            shared variable: x
    *** strlic: Error #2.4.17: "Exemple5_1.strl", line 11,
    col 16
```

```
        shared variable: x
*** strlic: Error #2.4.17: "Exemple5_1.strl", line 11,
col 35
        shared variable: x
*** strlic: 3 errors during compilation
*** esterel: exit: errors during strlic (1)
```

Exemple5_2.strl :

```
module Exemple5_2 :

  input  E1, E2, E3;
  output S1, S2, S3;

  await  E1;

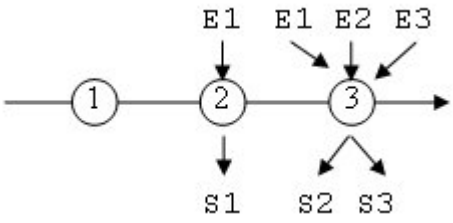
  [ emit S1; await  E1 || await E2; emit S2];

  emit S3

end module
```

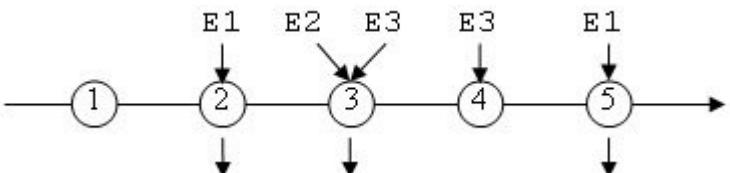
1er test :

```
Exemple5_2> ; E1; E1 E2 E3;.
--- Output:
--- Output: S1
--- Output: S2 S3
```



2ème test :

```
Exemple5_2> ; E1; E2 E3; E3; E1;.
--- Output:
--- Output: S1
--- Output: S2
--- Output:
--- Output: S3
```



6. **abort** : L'instruction permet l'abandon de l'exécution de son corps dès réception d'un signal donné. C'est une instruction retardée : les signaux de l'instance courante ne sont pas pris en compte.

Attention, ce n'est ni boucle, ni une instruction bloquante; l'instruction se termine par abandon ou quand son corps se termine.

```
abort
      corps
when  signal
```

```
abort
      corps
when  signal  do  corps_à_exécuter_si_signal_reçu  end abort
```

```
abort
      corps
when
   ...
   case  signal
   ...
   case  signal  do  corps_à_exécuter_si_signal_reçu
   ...
end abort
```

On peut mettre l'option **weak** devant le mot **abord** (penser dans ce cas à le mettre aussi dans **end weak abort**). Avec cette option, l'abandon se fait seulement après avoir exécuté toutes les instructions du corps qui ne sont pas ou plus retardées. Les instructions qui sont retardées ou qui se situent après les instructions retardées sont abandonnées (voir l'exemple Exemple6_3.strl ci-dessous).

Exemple6_1.strl :

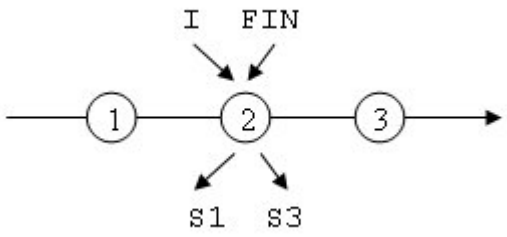
```
module Exemple6_1 :
  input I, FIN;
  output S1, S2, S3;

  await I;
  abort
    present FIN then emit S1 end present
  when FIN do emit S2
  end abort;
  emit S3

end module
```

1er test :

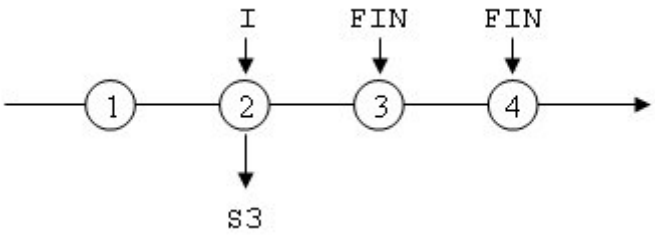
```
Exemple6_1> ; I FIN;;.
--- Output:
--- Output: S1 S3
--- Output:
```



- Dans le premier `tick`, comme `await` est une instruction retardée et bloquante, elle laisse passer tous les signaux émis (ici, 0 signal) dans ce `tick` pour attendre les signaux des `tick` suivants.
- Dans le `tick` suivant, les signaux I et FIN sont reçus ensemble.
- Comme `abort` est une instruction retardée, elle exécute son corps en laissant passer ce signal FIN pour attendre les signaux des `tick` suivants..
- Mais l'instruction `present` de son corps n'est pas retardée, teste ce signal pour émettre S1.
- Comme il n'y a rien qui attend dans le corps, le corps se termine, ce qui termine aussi l'instruction `abort`.

2ème test :

```
Exemple6_1> ; I; FIN; FIN;.
--- Output:
--- Output: S3
--- Output:
--- Output:
```



Comme il n'y a rien qui attend dans le corps, le corps se termine avant l'arrivée du signal FIN dans le prochain `tick`. Ce qui termine l'instruction `abort`. Les deux signaux FIN ne servent absolument à rien donc.

Exemple6_2.strl :

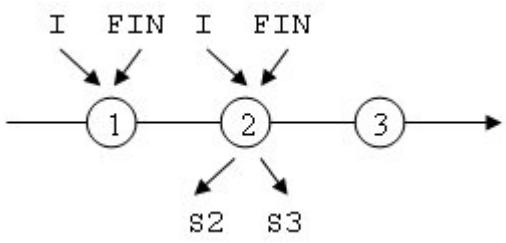
```
module Exemple6_2 :
  input I, FIN;
  output S0, S1, S2, S3;

  abort
    await I do emit S0 end await;
    present FIN then emit S1 end present
  when FIN do emit S2
  end abort;
  emit S3

end module
```

test :

```
Exemple6_1> I FIN; I FIN;;.
--- Output:
--- Output: S2 S3
--- Output:
```



- Dans le premier `tick`, les signaux I et FIN sont reçus.
- Comme `abort` est une instruction retardée, elle exécute son corps en laissant passer ces signaux.
- Mais dans son corps, l'instruction `await` est aussi retardée, donc laisse passer aussi ces signaux pour

attendre les signaux des prochains `tick` en bloquant l'exécution.

- Dans le prochain `tick`, les signaux I et FIN sont reçus. Comme `abort` a attendu le signal FIN, elle abandonne l'exécution de son corps (exécution qui s'est bloquée à `await I`). Ce qui explique que l'instruction `await` n'est pas exécutée. Si on veut terminer proprement toutes les instructions dépendantes de ce `tick` (voir l'exemple ci-dessous), il suffit d'utiliser l'option **`weak`**.

Exemple6_3.strl :

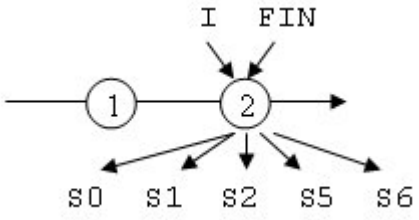
```
module Exemple6_3 :
  input I, J, FIN;
  output S0, S1, S2, S3, S4, S5, S6;

  weak abort
    await I do emit S0 end await;
    present FIN then emit S1 end present;
    emit S2;
    await J do emit S3 end await;
    emit S4
  when FIN do emit S5
  end weak abort;
  emit S6

end module
```

1er test :

```
Exemple6_3> ; I FIN;.
--- Output:
--- Output: S0 S1 S2 S5 S6
```

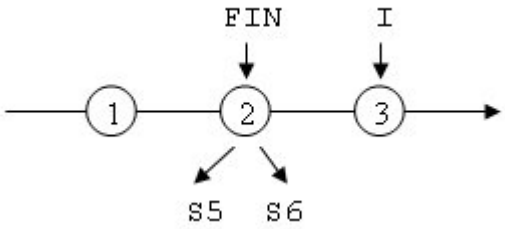


- Dans le premier `tick`, personne n'est intéressé, ni `abort`, ni `await I`. Tous les deux sont des instructions retardées. Elles attendent donc les signaux des prochains `tick`.
- Dans le deuxième `tick`,
  - les signaux I et FIN sont reçus ;
  - dans ce `tick`, `await I` … est satisfaite et émet S0;

- `present FIN` … qui n'est pas une instruction retardée est aussi satisfaite et émet S1;
- `emit S2` qui n'est pas une instruction retardée est aussi satisfaite et émet S2;
- `await J` … qui est une instruction retardée pour tester les signaux des prochains `tick` est abandonnée; pas d'émission de S3 ;
- les autres instructions qui suivent sont aussi abandonnées ; pas d'émission de S4, … ;

2ème test :

```
Exemple6_3> ; FIN; I;.
--- Output:
--- Output: S5 S6
--- Output:
```



- Dans le premier `tick`, personne n'est intéressé, ni `abort`, ni `await I`. Tous les deux sont des instructions retardées. Elles attendent donc les signaux des prochains `tick`.
- Dans le deuxième `tick`,
  - le signal FIN est reçu;
  - dans ce `tick`, `await I` … n'est pas satisfaite; elle attend encore les signaux des prochains `tick`, elle est donc encore retardée. Elle est donc abandonnée et les instructions qui suivent aussi.

7. **loop** et **every**

loop
corps
end loop

loop
corps
each *signal*

- Une instruction non bloquante permettant d'exécuter son corps en boucle.
- Pour la **`loop...end loop`**, lorsque l'exécution du corps est terminée, l'instruction répète dans le même `tick` l'exécution du corps.
  Inutile de définir ici un corps ne contenant que des instructions non bloquantes (comme par exemple ne contenant que des `emit`) en pensant pouvoir les faire répéter à l'infini dans le même `tick`; le compilateur `esterel` le détectera comme erreur.
- Pour la **`loop...each`**, c'est une instruction retardée : elle commence par exécuter son corps en ignorant les signaux reçus dans le `tick` courant (c'est-à-dire dans le `tick` où l'instruction commence son exécution). Si dans les `tick` suivants, le signal attendu arrive, **`loop...each`** abandonnera le cours de l'exécution de son corps pour recommencer à partir du début dans ce même `tick`.
- Pour la **`loop...each`**, lorsque l'exécution du corps est terminée, l'instruction ne recommencera l'exécution de son corps qu'au prochain *signal* reçu.

every *signal* **do** *a le même comportement que*
  await *signal*;
  corps
  loop *corps* **each** *signal*
end every

every immediate *signal* **do**
  corps
end every

- `every…end every` est une instruction retardée : elle ignore les signaux reçus dans le `tick` courant (c'est-à-dire dans le `tick` où l'instruction commence son exécution).
- Pour éviter ce retard et considérer les signaux reçus dans le `tick` courant, on utilise `every immediate…end every`.

Exemple7_1.strl :

```
module Exemple7_1 :
  input E1, E2;
  output S0, S1, S2;

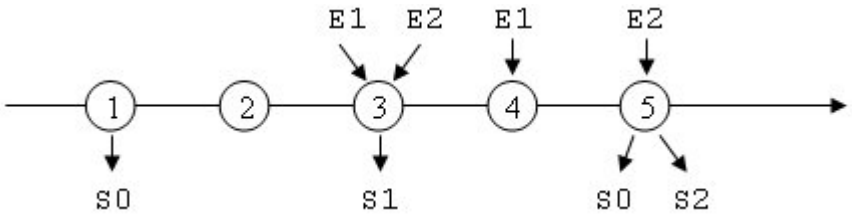
  loop
    emit S0;
```

```
    await E1;
    emit S1;
    await E2;
    emit S2
  end loop

end module
```

test :

```
Exemple7_1> ;; E1 E2; E1; E2;.
--- Output: S0
--- Output:
--- Output: S1
--- Output:
--- Output: S0 S2
```



- **Loop...end loop** est une instruction retardée, elle s'exécute en commençant par laisser tomber tous les signaux (ici, aucun signal) reçus dans son `tick` courant : ce qui permet d'émettre **S0**.
- Juste après l'émission et toujours dans le même `tick`, on exécute **await E1** qui est aussi une instruction retardée. Elle s'exécute en commençant par laisser tomber aussi tous les signaux reçus dans son `tick` courant. Mais elle est bloquante et restera bloquante jusqu'à la réception du signal **E1** dans les `tick` suivants.
- `tick` suivant, toujours rien.
- `tick` suivant, il y a deux signaux reçus **E1** et **E2** dont **E1** satisfait l'attente de **await E1**, ce qui permet d'émettre dans ce `tick` le signal **S1**.
- Juste après l'émission et toujours dans le même `tick`, on exécute **await E2** qui est une instruction retardée. Elle s'exécute en commençant par laisser tomber aussi tous les signaux reçus dans son `tick` courant. Mais elle est bloquante et restera bloquante jusqu'à la réception du signal **E2** dans les `tick` suivants.
- `tick` suivant, il y a le signal reçu **E1**. **await E2** n'est pas intéressée. Elle continue à attendre les signaux des prochains `tick`.

- `tick` suivant, arrive le signal **`E2`**. **`await E2`** est satisfaite. Dans ce même `tick`, **`S2`** est émis et c'est la fin du corps. **`loop...end loop`** reprend dans le même `tick` le début du corps. Ce qui permet l'émission du signal **`S0`** dans ce même `tick`.

Exemple7_2.strl :

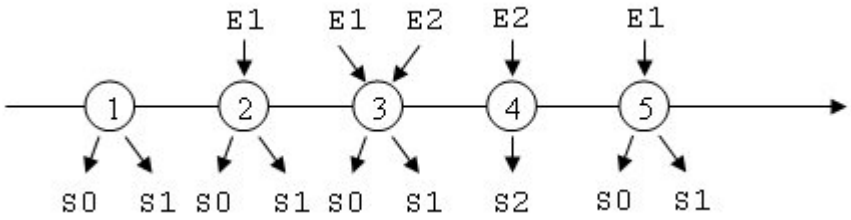
```
module Exemple7_2 :
  input E1, E2;
  output S0, S1, S2;

  loop
    emit S0;
    emit S1;
    await E2;
    emit S2;
  each E1

end module
```

test :

```
Exemple7_2> ; E1; E2 E1; E2; E1;.
--- Output: S0 S1
--- Output: S0 S1
--- Output: S0 S1
--- Output: S2
--- Output: S0 S1
```



- **`loop...each`** est une instruction retardée, elle s'exécute en commençant par laisser tomber tous les signaux (ici, aucun signal) reçus dans son `tick` courant. Ce qui permet d'émettre **`S0`** et **`S1`** dans ce `tick`. Si elle reçoit le signal **`E1`** dans les prochains `tick`, elle abandonnera son cours d'exécution pour reprendre l'exécution de son corps à partir du début.
- Juste après les émissions et toujours dans ce même `tick`, on exécute **`await E2`** qui est aussi une instruction retardée. Elle s'exécute en commençant par laisser tomber aussi tous les

signaux reçus dans son `tick` courant. Mais elle est bloquante et restera bloquante jusqu'à la réception du signal **`E2`** dans les `tick` suivants.

- `tick` suivant, le signal **`E1`** est reçu. Le signal **`E1`** permet donc à **`loop...each`** d'abandonner son cours d'exécution (ici, l'attente de **`await E2`**) pour recommencer au début du corps dans ce même `tick`. Ce qui permet d'émettre **`S0`** et **`S1`** dans ce même `tick`.
- Juste après les émissions et toujours dans ce même `tick`, on exécute **`await E2`**. Elle s'exécute en commençant par laisser tomber tous les signaux reçus dans son `tick` courant. Mais elle est bloquante et restera bloquante jusqu'à la réception du signal **`E2`** dans les `tick` suivants.
- `tick` suivant, les signaux **`E1`** et **`E2`** (l'ordre n'a aucune importance) sont reçus. Le signal **`E1`** permet donc à **`loop...each`** d'abandonner son cours d'exécution (ici, l'attente de **`await E2`**) pour recommencer au début du corps dans ce même `tick`. Ce qui permet d'émettre **`S0`** et **`S1`** dans ce même `tick`.
- Juste après les émissions et toujours dans ce même `tick`, on exécute **`await E2`**. Elle s'exécute en commençant par laisser tomber tous les signaux reçus dans son `tick` courant. Mais elle est bloquante et restera bloquante jusqu'à la réception du signal **`E2`** dans les `tick` suivants.
- `tick` suivant, le signal **`E2`** est reçu. Le signal **`E2`** permet donc de satisfaire **`await E2`** dans ce `tick`. Ce qui permet donc d'émettre **`S2`** dans ce `tick`.
- C'est aussi la fin du corps. **`loop...each`** attend le prochain **`E1`** pour recommencer le corps.

Exemple7_3.strl :

```
module Exemple7_3 :
  input E;
  output S1, S2;

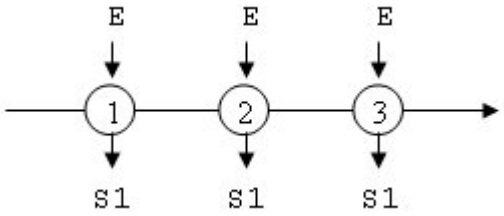
  loop
    emit S1;
    await E;
    emit S2;
  each tick

end module
```

test :

```
Exemple7_3> E; E; E;.
--- Output: S1
```

```
--- Output: S1
--- Output: S1
```



- **`loop...each`** est une instruction retardée, elle s'exécute en commençant par laisser tomber tous les signaux (ici, le premier **`E`**) reçus dans son `tick` courant. Ce qui permet d'émettre **`S1`** dans ce `tick`.
- Juste après l'émission et toujours dans ce même `tick`, on exécute **`await E`** qui est aussi une instruction retardée. Elle s'exécute en commençant par laisser tomber aussi tous les signaux reçus dans son `tick` courant (ici, toujours le premier **`E`**). Mais elle est bloquante et restera bloquante jusqu'à la réception du signal **`E`** dans les `tick` suivants.
- Au `tick` suivant, comme son nom l'indique, c'est le signal `tick`. On recommence le corps.
- Aucune chance de satisfaire **`await E`**, donc aucune chance d'émettre **`S2`**.

Exemple7_4.strl :

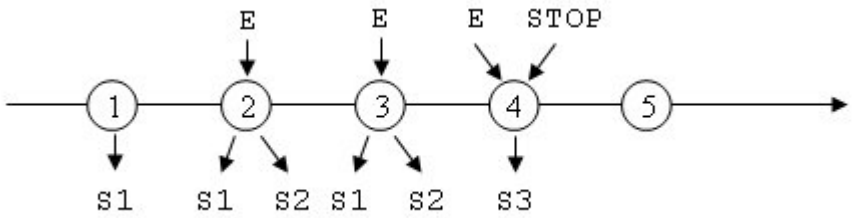
```
module Exemple7_4 :
  input E, STOP;
  output S1, S2, S3;

  abort
    loop
      emit S1;
      await E;
      emit S2;
    end loop
  when STOP do emit S3 end abort

end module
```

test :

```
Exemple7_4> ; E; E; E STOP;;.
--- Output: S1
--- Output: S1 S2
--- Output: S1 S2
--- Output: S3
--- Output:
```



Exemple7_5.strl :

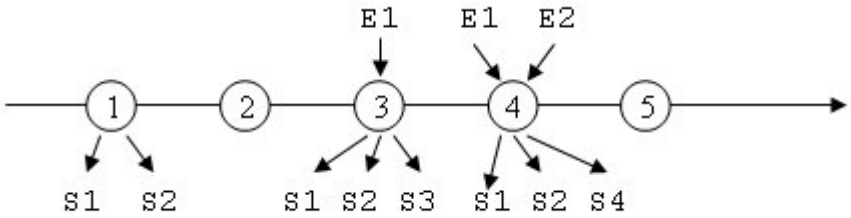
```
module Exemple7_5 :
  input E1, E2;
  output S1, S2, S3, S4;

  loop
    emit S1;
    abort
      emit S2;
      await E1;
      emit S3
    when E2 do emit S4 end abort
  end loop

end module
```

test :

```
Exemple7_5> ;; E1; E1 E2;;.
--- Output: S1 S2
--- Output:
--- Output: S1 S2 S3
--- Output: S1 S2 S4
--- Output:
```



Exemple7_6.strl :

```
module Exemple7_6 :
  input I;
  output O1, O2, O3, O4;

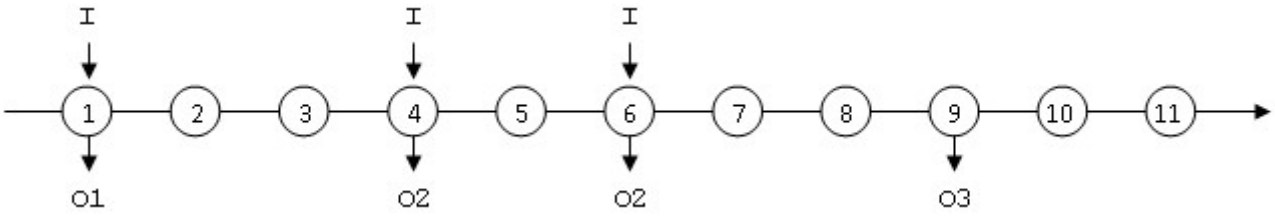
  emit O1;
  every I do
    emit O2;
    await 3 tick;
```

```
    emit O3;
  end every;
  emit O4

end module
```

test :

```
Exemple7_6> I;;;I;;I;;;;;;.
--- Output: O1
--- Output:
--- Output:
--- Output: O2
--- Output:
--- Output: O2
--- Output:
--- Output:
--- Output: O3
--- Output:
--- Output:
```



Exemple7_7.strl :

```
module Exemple7_7 :
  input I;
  output O1, O2, O3, O4;

  emit O1;

  every immediate I do
    emit O2;
    await 3 tick;
    emit O3;
  end every;

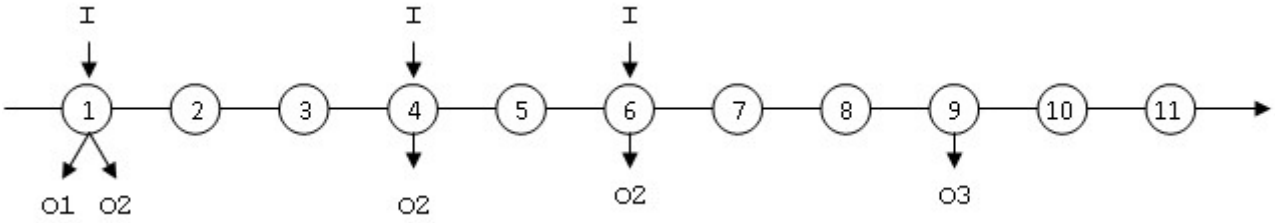
  emit O4;

end module
```

test :

```
Exemple7_7> I;;;I;;I;;;;;;
--- Output: O1 O2
--- Output:
```

```
--- Output:
--- Output: O2
--- Output:
--- Output: O2
--- Output:
--- Output:
--- Output: O3
--- Output:
--- Output:
```



8. sustain

sustain *signal*

- Une instruction bloquante permettant d'émettre sans attendre un signal à chaque `tick` à commencer par le `tick` courant.

Exemple8_1.strl :

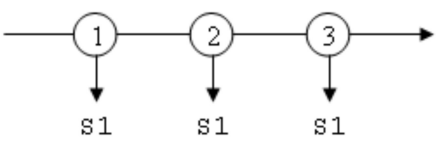
```
module Exemple8_1 :
  output S1, S2;

  sustain S1;              % bloquante
  emit S2;                 % ne passe donc plus ici

end module
```

test :

```
Exemple8_1> ;;;.
--- Output: S1
--- Output: S1
--- Output: S1
```



Exemple8_2.strl :

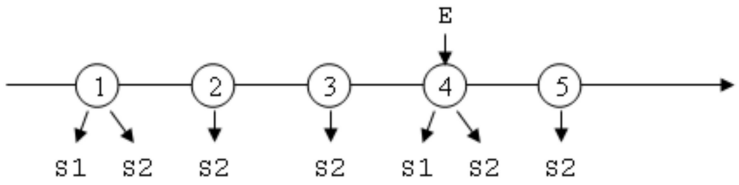
```
module Exemple8_2 :
  input E;
  output S1, S2, S3;

  loop
    emit S1;
    sustain S2;
    emit S3;        % ne passe toujours pas par ici
  each E;

end module
```

test :

```
Exemple8_2> ;;;E;;.
--- Output: S1 S2
--- Output: S2
--- Output: S2
--- Output: S1 S2
--- Output: S2
```



9. **suspend** instruction retardée (les signaux de l'instant courant ne sont pas pris en compte). Elle commence donc à exécuter le corps. Elle permet de suspendre l'exécution du corps :

   Au cours de l'exécution des instructions du corps, si le *signal* apparaît, l'exécution des instructions correspondant à ce `tick` (où apparaît ce *signal*) est suspendue. Dans un prochain `tick` où le signal disparaîtra, l'exécution reprendra à l'endroit exact où elle est suspendue.

suspend
    corps
when *signal*

Exemple9_1.strl :

```
module Exemple9_1 :
  input  E1, E2;
  output S1, S2;

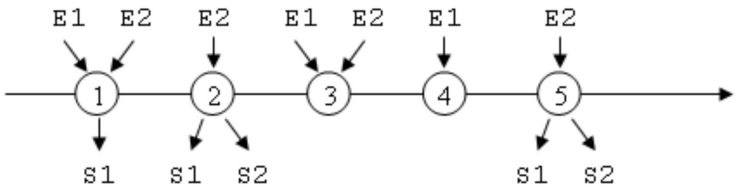
  suspend
    loop
```

```
      emit S1;
      await E2;
      emit S2;
    end loop
  when E1

end module
```

test :

```
Exemple9_1> E1 E2; E2; E1 E2; E1; E2;.
--- Output: S1
--- Output: S1 S2
--- Output:
--- Output:
--- Output: S1 S2
```



Exemple9_2.strl :

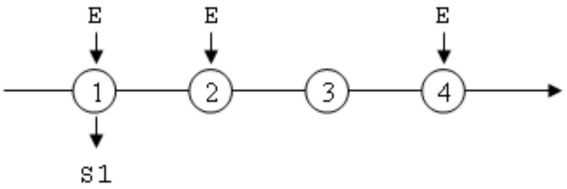
```
module Exemple9_2 :
  input  E;
  output S1, S2;

  suspend
    loop
      emit S1;
      await E;
      emit S2;      % ne passera jamais par ici
    end loop
  when E

end module
```

test :

```
Exemple9_2> E; E;; E;.
--- Output: S1
--- Output:
--- Output:
--- Output:
```



10. **trap** instruction permettant de faire des échappements par la levée de ses exceptions concurrentes.

L'instruction se termine quand son corps se termine ou par abandon lorsque une ou plusieurs de ses exceptions concurrentes sont levées par la commande **exit** *exception*. Dans ce dernier cas,

- les instructions du corps derrières ces **exit** sont abandonnées,
- les instructions qui sont retardées ou qui se situent après les instructions retardées sont abandonnées,
- les instructions du corps qui ne sont pas ou plus retardées sont exécutées (voir **weak abort**),
- et pour les exceptions gérées (**handle**), l'abandon s'opère après avoir exécuté en parallèle tous les $corps_j$, … et $corps_k$ correspondants respectivement aux exceptions $exception_1$, …et $exception_k$ levées en concurrence (dans le même `tick`).

trap $exception_1$, …, $exception_n$ **in** % déclaration des exceptions concurrentes
    corps
end trap

trap $exception_1$, …, $exception_n$ **in**
    corps
    handle $exception_j$ **do** $corps_j$
    …
    handle $exception_k$ **do** $corps_k$
    …
end trap

Exemple10_1.strl :

```
module Exemple10_1 :
  input  E;
  output S1_1, S1_2, S2_1, S2_2, S3, S4, S5;

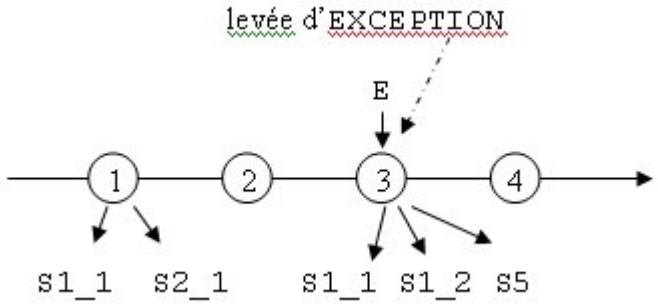
  trap EXCEPTION in
    [ loop
        emit S1_1;
        await E;
```

```
        emit S1_2;
      end loop;
      emit S3
      ||
      emit S2_1;
      await E;
      exit EXCEPTION;
      emit S2_2
    ];
    emit S4
  end trap;
  emit S5;

end module
```

test :

```
Exemple10_1> ;;E;;.
--- Output: S1_1 S2_1
--- Output:
--- Output: S1_1 S1_2 S5
--- Output:
```



- Rappel : l'opérateur séquence `;` est prioritaire par rapport à l'opérateur `||`.
  Nous avons donc en parallèle, les blocs

  ```
  loop … end loop ; emit S3
  ```

  et

  ```
  emit S2_1 ; …; exit EXCEPTION; emit S2_2
  ```

- Au 1er `tick`, les signaux `S1_1` et `S2_1` sont émis. Tous les deux sont bloqués en attente du signal `E`.
- Au 2ème `tick`, pas de signal `E`. Ils sont toujours bloqués.
- Au 3ème `tick`, le signal `E` est reçu par les deux attentes parallèles.
  - Dans le 2ème bloc, il y a levée d'`EXCEPTION` et les instructions qui suivent ne seront jamais exécutées (Ici, `emit S2_2`, mais aussi dans une autre mesure, `emit S4`, car cette dernière ne sera exécutée que si les blocs en parallèle sont terminés. Ce qui n'est pas le cas ici).

- Dans le premier bloc, il y a émission de `S1_2`. Comme il n'y a pas encore d'instruction retardée, on continue la boucle. On émet `S1_1`, puis on tombe sur l'instruction retardée `await E`.
- Il ne reste donc que les instructions retardées ou derrières les instructions retardées ou derrières l'`exit`. On abandonne le corps du `trap`.

Exemple10_2.strl :

```
module Exemple10_2 :
  input  E, E1, E2, E3;
  output S, S1, S3, S4;

  trap X3, X1, X2 in

    loop
      emit S;
      await E;
    end loop
    ||
    await E1; exit X1
    ||
    await E2; exit X2
    ||
    await E3; exit X3

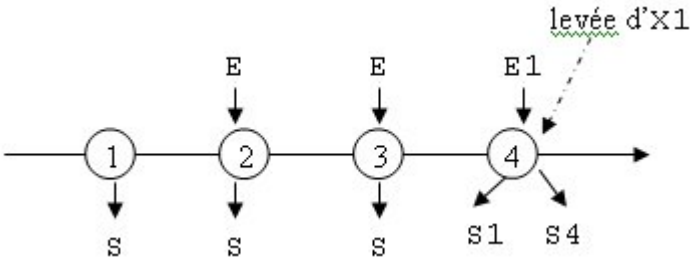
    handle X1 do emit S1
    handle X3 do emit S3
  end trap;

  emit S4

end module
```

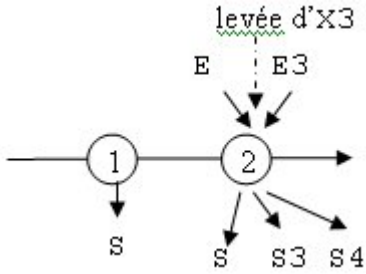
test :

```
Exemple10_2> ;E;E;E1;.
--- Output: S
--- Output: S
--- Output: S
--- Output: S1 S4
```



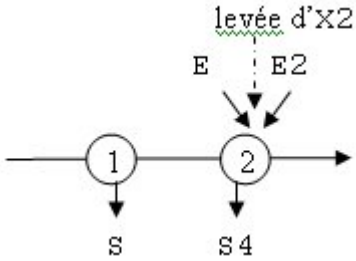
test :

```
Exemple10_2> ;E E3;.
--- Output: S
--- Output: S S3 S4
```



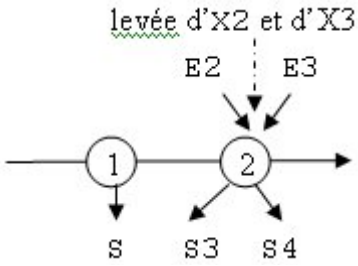
test :

```
Exemple10_2> ;E2;.
--- Output: S
--- Output: S4
```



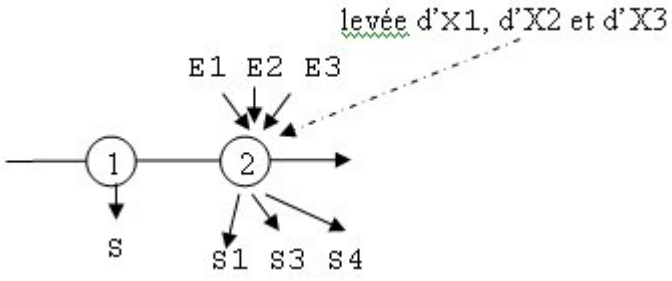
test :

```
Exemple10_2> ;E2 E3;.
--- Output: S
--- Output: S3 S4
```



test :

```
Exemple10_2> ;E1 E2 E3;.
--- Output: S
--- Output: S1 S3 S4
```



11. Appel des fonctions et procédures extérieures

- On peut appeler des fonctions ou procédures externes (ici, écrites en C).
- On doit d'abord déclarer leurs signatures dans la partie « Déclarations d'interface » du module.
- Signatures :
  - **function** nom **(**$type_1$**,...,** $type_n$**)** **:** type_retour **;**
  - **procedure** nom **(**$type_1$**,...,** $type_m$**)** **(**$type_1$**,...,** $type_n$**);**
- Il y a deux listes d'arguments pour les procédures : la 1ère concerne des arguments passés par référence et la 2ème des arguments passés par valeur.
- On appelle une fonction par son nom et une procédure par son nom précédé d'un **call**.
- Le fichier `.c` généré par Esterel a besoin d'un fichier `.h` (à créer) de même nom.
- On créée ce fichier `.h` avec les prototypes des fonctions ou procédures utilisées et on l'accompagne avec un fichier `.c` avec leurs définitions. Ou tout simplement, on créée ce fichier `.h` avec leurs définitions.

Exemple11_1.strl :

```
module Exemple11_1 :

  input  FACT := 0 : integer;
  output RESULTAT : integer;

  function factoriel (integer) : integer;

  loop
    present FACT
      then emit RESULTAT(factoriel(?FACT))
    end present;
    await FACT
  end loop

end module
```

avec Exemple11_1.h :

```
int factoriel (int n) {
  if (n == 0)
    return 1;
  else return n * factoriel(n - 1);
}
```

test :

```
Exemple11_1> FACT(3);;FACT(5);.
--- Output: RESULTAT(6)
--- Output:
--- Output: RESULTAT(120)
```

Exemple11_2.strl :

```
module Exemple11_2 :

  input  FACT := 0 : integer;
  output RESULTAT : integer;

  procedure factoriel (integer) (integer);

  loop
    present FACT
      then
        var aux := 1 : integer in
          call factoriel (aux) (?FACT);
          emit RESULTAT(aux)
        end var
    end present;
    await FACT
  end loop

end module
```

avec Exemple11_2.h :

```
void factoriel (int *x, int n) {
  int i;

  for (i = 1; i <= n; ++i) {
    *x *= i;
  }
}
```

test :

```
Exemple11_2> FACT(3);;FACT(5);.
--- Output: RESULTAT(6)
--- Output:
--- Output: RESULTAT(120)
```

12. combine

- Dans un `tick`, il est possible qu'un signal est émis plusieurs fois, mais un seul (le dernier) est présent et réellement émis (voir Exemple2_1.strl ci-dessus).
- Le problème peut se poser lorsque le signal est valué.
- Si on déclare un signal valué en utilisant `combine` muni d'un opérateur (ou fonction) binaire, commutatif et associatif, les différentes valeurs du signal émis dans le même `tick` seront combinées en utilisant cet opérateur (ou fonction).
- `combine` permet entre autre de répondre aux problèmes posés par l'interdiction d'utiliser une variable dans la composition parallèle des instructions (voir Opérateur parallèle ||).

Exemple12_1.strl :

```
     module Exemple12_1 :

  output S1 : integer;
  signal S2 : integer in

    await S2; emit S1(?S2)
    ||
    pause; emit S2(4)
    ||
    pause; emit S2(3)
    ||
    pause; emit S2(2)
    ||
    pause; emit S2(1)

  end signal

end module
```

test :

```
Exemple12_1> ;;;;.
--- Output:
--- Output: S1(1)
--- Output:
--- Output:
```

Exemple12_2.strl :

```
     module Exemple12_2 :

       output S1 : integer;
       signal S2 := 0 : combine integer with + in
         var accumulateur := 0 : integer in
           loop
```

```
        await S2; accumulateur := accumulateur + ?S2;
emit S1(accumulateur)
        ||
        pause; emit S2(1)
        ||
        pause; emit S2(2)
        ||
        pause; emit S2(3)
        ||
        pause; emit S2(4)
      end loop
    end var
  end signal

end module
```

test :

```
Exemple12_2> ;;;;.
--- Output:
--- Output: S1(10)
--- Output: S1(20)
--- Output: S1(30)
```

Exemple12_3.strl :

```
module Exemple12_3 :

  output S1 : integer,  S2 := 0 : combine integer with +;

  var accumulateur := 0 : integer in
    loop
      await S2; accumulateur := accumulateur + ?S2; emit
S1(accumulateur)
      ||
      pause; emit S2(1)
      ||
      pause; emit S2(2)
      ||
      pause; emit S2(3)
      ||
      pause; emit S2(4)
    end loop
  end var

end module
```

test :

```
Exemple12_3> ;;;;.
--- Output:
--- Output: S1(10) S2(10)
--- Output: S1(20) S2(10)
```

```
--- Output: S1(30) S2(10)
```

Exemple12_4.strl :

```
module Exemple12_4 :

  function max (integer, integer) : integer;

  output S1 : integer,  S2 := 0 : combine integer with
max;

  var accumulateur := 0 : integer in
    loop
      await S2; accumulateur := accumulateur + ?S2; emit
S1(accumulateur)
      ||
      pause; emit S2(1)
      ||
      pause; emit S2(2)
      ||
      pause; emit S2(3)
      ||
      pause; emit S2(4)
    end loop
  end var

end module
```

Avec Exemple12_4.h :

```
int max (int x, int y);
```

Avec max.c :

```
int
max (int x, int y)
{
  return (x > y ? x : y);
}
```

test :

```
Exemple12_4> ;;;;.
--- Output:
--- Output: S1(4) S2(4)
--- Output: S1(8) S2(4)
--- Output: S1(12) S2(4)
```

13. if

- **if** *expression booléenne* **then** *instructions* **else** *instructions* **end if**
- **if** *expression booléenne* **then** *instructions* **end if**
- **if** *expression booléenne* **else** *instructions* **end if**
- **if** *expression booléenne* **then** *instructions*
  elsif *expression booléenne* **then** *instructions*
  ...
  elsif *expression booléenne* **then** *instructions*
  else *instructions* **end if**

Exemple13.strl :

```
module Exemple13 :

  input  E : integer, STOP;
  output S : integer, FIN;

  abort
    loop
      await E;
      if ?E mod 2 = 0
         then emit S(2)
      elsif ?E mod 3 = 0
         then emit S(3)
      elsif ?E mod 5 = 0
         then emit S(5)
      else emit S(0)
      end if
    end loop
  when STOP do emit FIN end abort

end module
```

test :

```
Exemple13> ;; E(2) ; E(3); E(4);; E(5); E(6) STOP;;.
--- Output:
--- Output:
--- Output: S(2)
--- Output: S(3)
--- Output: S(2)
--- Output:
--- Output: S(5)
--- Output: FIN
--- Output:
```

14. Quelques erreurs à éviter

Exemple14_1.strl :

```
module Exemple14_1 :
input E1, E2, E3;
output S1, S2, S3;

% Erreur, car il y a un ordre (de sequences) dans
l'instant
% [
%    await immediate E1;
%    emit E1;
%    emit S1
% ]
% ||


[
   emit E1;
   await immediate E1;
   emit S1
]
||
[
   emit E2;
   await immediate E3;
   emit S2
]
|||[
   emit E3;
   await immediate E2;
   emit S3
]

end module
```

test :

```
Exemple14_1> ;;;
--- Output: S1 S2 S3
--- Output:
--- Output:
Exemple14_1> .
```

Exemple14_2.strl :

```
module Exemple14_2 :
input E1, E2, E3, E4;
output S1, S2, S3, S4;

% Erreur : on ne peut pas emettre un signal de son corps
%    dans l'instant ou on attend ce meme signal.
% [
%    abort
%       pause;
```

```
%       emit E1;
%    when E1;
%    emit S1
% ]
%
% ||
%
% Erreur : meme situation, mais croisee.
% [
%   abort
%      pause;
%      emit E2;
%   when E3;
%    emit S2
% ]
%
% ||
%
% [
%    abort
%       pause;
%       emit E3;
%    when E2;
%    emit S3
% ]

[
   abort
      await tick;
      emit E1;
      await 100 tick;
   when E4;
   emit S1;
]

||

[
   weak abort
      await E1;
      emit E2;
      await 100 tick;
   when E1;
   emit S2
]

||

[
   weak abort
      await E2;
      emit S3;
      await 100 tick;
   when E2;
   emit S4
```

```
]

end module
```

test :

```
Exemple14_2> ;;;E4;;
--- Output:
--- Output: S2 S3 S4
--- Output:
--- Output: S1
--- Output:
Exemple14_2> .
```