

# Systemes et langages réactifs synchrones

Nadine Richard - Nadine.Richard@enst.fr

## 1 Introduction

Selon D. Harel et A. Pnueli, les systèmes informatiques peuvent être catégorisés comment étant réactifs, interactifs ou transformationnels, en fonction de leur degré d'interaction avec leur environnement [16]. Contrairement à un système transformationnel, correspondant à un unique calcul de fonction, un système interactif ou un système réactif doivent maintenir une interaction constante avec leur environnement. Nous verrons dans ce chapitre les différentes solutions actuellement proposées pour concevoir des systèmes réactifs, en particulier lorsque ceux-ci doivent respecter des contraintes de temps-réel. Nous présenterons à ce propos le modèle synchrone, basé sur la notion d'instant et sur l'hypothèse d'un temps nul de réaction du système. Nous nous intéresserons tout particulièrement au langage impératif synchrone ESTEREL. Nous étudierons ensuite la mise en œuvre d'un système synchrone, qui passe par la conception d'une machine d'exécution regroupant les mécanismes d'interfaçage nécessaires entre le système et son environnement asynchrone. Avant de conclure, nous présenterons deux modèles d'entités synchrones : les **réseaux de processus réactifs** et les **objets synchrones**.

## 2 Approche synchrone pour les systèmes réactifs

### 2.1 Systèmes transformationnels, interactifs et réactifs

Historiquement, les systèmes transformationnels et interactifs proviennent de la programmation classique, à laquelle ont été ajoutés des mécanismes de gestion d'événements, de synchronisation et de programmation concurrente, alors que les systèmes réactifs sont issus des domaines du génie électrique, de l'automatique et des systèmes embarqués. Un **système transformationnel** effectue des calculs à partir des données fournies en entrée, pour produire des résultats en sortie avant de se terminer (*cf.* figure 1). L'interaction avec l'environnement se limite donc à l'acquisition des données et à la production de résultats, comme dans le cas d'un compilateur par exemple. Un tel système n'a pas d'état interne, le résultat dépend ainsi uniquement des données en entrée (à moins d'utiliser des fonctions aléatoires).

Les **systèmes interactifs** et **réactifs** interagissent continuellement avec leur environnement, en produisant des résultats à chaque invocation. Ces résultats dépendent des données fournies par l'environnement lors de l'invocation, ainsi que de l'état interne du système. La différence entre ces deux types de système réside dans l'entité qui contrôle l'interaction. Dans un système interactif comme une base de données, la prise en compte des requêtes et la production de réponses se font à l'initiative du système,

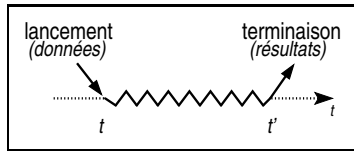


FIG. 1 – Exécution d'un système de type transformationnel.

qui impose ainsi son propre rythme comme le montre la figure 2.

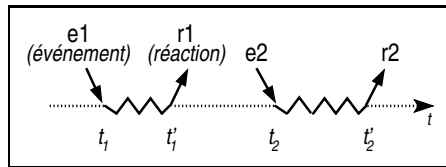


FIG. 2 – Exécution d'un système de type interactif.

Un système réactif doit, quant à lui, être toujours en mesure de fournir une réponse immédiate quand l'environnement le sollicite. L'évolution d'un système réactif est donc une suite de réactions provoquées par l'environnement, chaque réaction étant considérée comme instantanée par rapport à l'échelle de temps propre à l'environnement (*cf.* figure 3). Les interfaces homme-machine et les programmes de contrôle de processus industriels sont des exemples typiques de systèmes réactifs. La notion de réactivité est bien sûr une représentation idéalisée de systèmes qui, vis-à-vis de leur environnement, doivent réagir de manière réflexe. En pratique, la plupart des systèmes réactifs sont implémentés par des systèmes interactifs suffisamment rapides pour prendre en compte tous les stimuli et y répondre à temps.

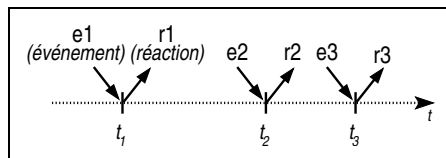


FIG. 3 – Exécution d'un système de type réactif.

En pratique, rares sont les applications purement réactives : un grand nombre d'applications sont constituées d'un cœur réactif et de traitements transformationnels s'exécutant en parallèle. Le distributeur automatique de billets décrit dans [6] est un exemple représentatif de système composé de parties réactives, interactives et transformationnelles.

## 2.2 Systèmes réactifs temps-réel

Les systèmes réactifs sont en particulier utilisés dans le cadre des systèmes temps-réel. Le terme **temps-réel** (*real-time*) est souvent utilisé sous sa forme galvaudée pour qualifier une application «qui réagit rapidement aux événements», c'est-à-dire une application interactive avec un temps de réponse acceptable pour l'utilisateur. Il est aussi

utilisé dans le cadre de simulations ou d'animations calculées pendant leur visualisation, c'est-à-dire là encore pour des applications interactives à faible temps de réponse. Dans un véritable système temps-réel, les temps de réponse ne constituent pas seulement des mesures de performance, mais sont des contraintes fortes<sup>1</sup> : le système doit réagir aux événements dans des délais prédéfinis. Il fonctionne correctement s'il fournit le résultat attendu et si ce résultat arrive «dans les temps» : un bon résultat produit trop tard est considéré comme inutile.

Généralement, un système temps-réel possède par ailleurs deux caractéristiques fondamentales, en particulier s'il joue un rôle critique c'est-à-dire s'il met en jeu des vies humaines (*safety critical*) ou si la réussite de sa mission est primordiale (*mission critical*) :

- la prévisibilité : il faut pouvoir prévoir le comportement du système, qui doit donc être parfaitement **déterministe**. Un système est dit déterministe s'il produit toujours exactement la même séquence de résultats à partir d'une même séquence de données ou d'événements ;
- la sûreté de fonctionnement : le comportement du système doit pouvoir être garanti même dans des conditions extrêmes (fortes variations de température, vibrations, *etc.*), qui augmentent les risques de panne des composants.

Les systèmes temps-réel dits embarqués, réactifs ou non, sont utilisés en particulier dans des véhicules, par exemple pour le pilotage automatique d'avions : la prévisibilité et la sûreté de fonctionnement sont alors évidemment des priorités. Les systèmes réactifs se doivent d'être déterministes et sûrs lorsqu'ils sont utilisés pour des applications temps-réel critiques.

## 2.3 Approches traditionnelles

Avant que le modèle synchrone soit proposé, les approches les plus couramment rencontrées pour implémenter des systèmes réactifs étaient les automates, les langages de haut-niveau associés à des exécutifs temps-réel multi-tâches et les langages parallèles asynchrones. Programmer un système directement sous la forme d'un automate permet d'obtenir un comportement déterministe et, en général, une exécution efficace ; de plus, de nombreux outils permettent de vérifier formellement le comportement de l'automate. Les automates sont hélas difficiles à maintenir. Cette maintenabilité est cependant améliorée par les formalismes étendus de description d'automates comme HPTS, qui intègrent le parallélisme et la hiérarchisation ; les automates ainsi décrits sont généralement traduits en automates séquentiels.

Le système peut aussi être conçu comme un ensemble de tâches séquentielles co-opérantes<sup>2</sup>, qui communiquent de façon asynchrone. Dans le modèle de communication asynchrone, l'appelant peut continuer de s'exécuter après avoir envoyé une requête, en attendant que celle-ci soit traitée par l'appelé. Bien qu'efficace à l'exécution, une solution à base de tâches séquentielles souffre de l'asynchronisme entre les tâches,

---

<sup>1</sup>Des contraintes temporelles strictes sont appliquées aux systèmes temps-réel dits «durs» (*hard real-time*) ; il s'agit typiquement d'un temps maximal entre l'occurrence d'un événement et la réponse du système. Ces contraintes sont assouplies pour les systèmes à contraintes temporelles relatives (*soft real-time*), qui doivent respecter des temps de réponse moyens.

<sup>2</sup>Dans le modèle d'ordonnancement coopératif, la tâche en cours d'exécution peut choisir de donner la main à une autre tâche, quelle que soit sa priorité. Il s'oppose au modèle préemptif, dans lequel une tâche plus prioritaire peut prendre la main sans tenir compte des besoins de la tâche courante.

qui rend délicates l'analyse et la mise au point d'un comportement global déterministe. De plus, les inversions de priorités dues au modèle coopératif favorise de manquement des échéances. Dans un langage comme ADA, le parallélisme et la synchronisation sont exprimées à l'aide de primitives spécifiques (tâches et rendez-vous). Un tel langage est assurément un progrès en ce qui concerne le confort de programmation, mais le problème fondamental de l'asynchronisme subsiste. Dans le cas des tâches coopérantes ou d'un langage comme ADA, il est nécessaire de disposer d'un exécutif multi-tâches, qui fournit les services nécessaires au parallélisme, à la communication et à la synchronisation entre les tâches. Cet exécutif peut par ailleurs lui-même utiliser les services d'un système d'exploitation temps-réel comme CHORUS ou ECOS.

Le développement d'une application temps-réel réactive doit être un processus rigoureux, nécessitant des outils adaptés, en particulier des langages pour la spécification et des outils fiables pour la vérification automatique. Selon H. Boufaïed [6], les solutions traditionnelles n'apportant pas de solution satisfaisante à la programmation de systèmes réactifs temps-réel, il a été proposé de développer des langages dédiés plutôt que d'adapter des langages existants. Cela devait permettre de prendre directement en compte les spécificités de ces systèmes à un haut niveau d'abstraction, en particulier le déterminisme et le contrôle fin des processus réactifs, malgré la complexité des comportements décrits. Le **modèle synchrone** a alors été introduit et de nombreux langages sont maintenant basés sur cette approche.

## 2.4 Approche réactive synchrone

L'hypothèse synchrone définit une échelle de temps logique discret, constituée d'**instants** correspondant à chacune des **réactions** du système. Les événements ayant déclenché la réaction sont considérés comme **simultanés**. De plus, les réactions se font en **temps nul** : les signaux émis pendant une réaction sont donc simultanés avec les signaux qui ont provoqué la réaction (la production de la réponse a lieu au même instant logique). Une réaction est donc par construction instantanée, ce qui évite les réactions concurrentes partielles d'un système, sources d'indéterminisme. La figure 4 illustre cette propriété : plusieurs événements sont pris en compte à chaque réaction, pour produire instantanément des résultats.

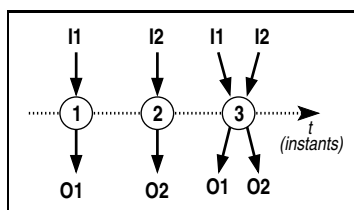


FIG. 4 – Un exemple d'exécution d'un système synchrone.

Les hypothèses synchrones conduisent à une vision abstraite des interactions et permettent de simplifier l'expression des comportements réactifs. Les fondements théoriques des langages synchrones garantissent par ailleurs des propriétés remarquables aux constructions de haut-niveau<sup>3</sup>. La composition parallèle d'ESTEREL est par exem-

<sup>3</sup>voir à ce propos l'article de G. Berry sur les fondements d'ESTEREL [2]

ple parfaitement déterministe, alors que la concurrence dans les langages asynchrones introduit généralement de l'indéterminisme. Les compilateurs s'appuient sur la sémantique des langages synchrones pour produire du code déterministe et efficace à l'exécution, et pour vérifier certaines propriétés en détectant par exemple les situations d'interblocage. Ces sémantiques mathématiques rigoureuses ont par ailleurs permis de construire des outils automatiques de vérification formelle, permettant de tester les programmes avant de les exécuter.

De plus, les réactions se font en temps nul, ce qui signifie simplement que toute réaction commencée doit terminer avant qu'une autre réaction soit déclenchée. La principale conséquence de ces hypothèses est la possibilité de composer des systèmes synchrones pour obtenir d'autres systèmes synchrones, le comportement résultant étant parfaitement défini et déterministe. Nous verrons plus tard que cette hypothèse est tout à fait acceptable au niveau du modèle, mais qu'elle pose des problèmes lors de l'implémentation (*cf.* section 4, page 19).

## 2.5 Langages et outils

L'hypothèse synchrone est à la base de plusieurs langages, tels que LUSTRE [15], ESTEREL [4, 3] et SIGNAL [10]. Les formalismes SYNCCHARTS [1] et HPTS [18] constituent des outils expressifs pour la spécification explicite d'automates selon le modèle synchrone. SYNCCHARTS repose sur une sémantique proche de celle d'ESTEREL. SIGNALGTI est une extension de SIGNAL intégrant en particulier la préemption hiérarchique de tâches<sup>4</sup> ; il a été utilisé par S. Donikian pour implémenter les HPTS.

Des langages synchrones comme LUSTRE ou SIGNAL, ainsi que le modèle HPTS, sont bien adaptés aux applications réactives manipulant essentiellement des flots de données (signaux continus) sous la forme de systèmes d'équations. ESTEREL et le formalisme SYNCCHARTS sont pour leur part utilisés pour des applications réactives pour lesquelles le contrôle joue un rôle primordial, c'est-à-dire qui doivent principalement réagir à des événements (signaux discrets). Nous nous intéressons à ce dernier type de langages synchrones et plus particulièrement à ESTEREL, qui est un langage de nature impérative adapté à la programmation événementielle.

Basé sur les états et les transitions, le formalisme SYNCCHARTS intègre les caractéristiques principales d'ESTEREL : la hiérarchie par l'imbrication de macro-états, la concurrence par la description de graphes d'états indépendants dans le même macro-état, la communication par diffusion instantanée de signaux, ainsi que la préemption (avortement et suspension). Un SYNCCHART peut donc être traduit automatiquement en un programme ESTEREL. Un éditeur graphique de SYNCCHART a été développé par C. André, complétant ainsi l'atelier de conception de systèmes réactifs dédié à ESTEREL constitué d'un compilateur et d'outils de validation formelle [1].

## 3 Le langage ESTEREL

ESTEREL est un langage réactif synchrone de nature impérative, basé d'une part sur des structures de contrôle comme l'itération, la séquence et la composition parallèle déterministe, et d'autre part sur un ensemble d'instructions dites réactives faisant

---

<sup>4</sup>GTI signifie Gestion de Tâches et intervalles.

référence aux événements auxquels le système décrit doit réagir. La communication et la synchronisation entre les différentes composantes d'un programme ESTEREL se fait par diffusion instantanée de signaux correspondant à ces événements. Le langage offre par ailleurs des facilités de manipulation de données, de programmation modulaire et de gestion de tâches asynchrones, ainsi que des mécanismes de gestion d'exceptions et de préemption. Une présentation complète du langage est disponible dans [3]; nous présenterons ici les notions et instructions caractéristiques d'ESTEREL.

### 3.1 Principes du langage

Un **signal** est caractérisé par son **statut**, c'est-à-dire par sa présence ou son absence au cours d'un instant. Il peut être émis par l'environnement si c'est un signal en entrée, ou par le programme ESTEREL avec l'instruction `emit` si c'est un signal en sortie. Les signaux sont diffusés instantanément et sont visibles dans toutes les composantes parallèles du programme : un signal émis par une composante est reçu par toutes les autres composantes dans le même instant. Une instruction débute à un instant  $t$ , appelé le «premier instant» pour cette instruction ; l'instruction peut rester active pendant plusieurs instants avant de se terminer à un instant  $t'$  tel que  $t' \geq t$ . Une instruction est dite **instantanée** si  $t = t'$ . Une instruction se termine soit spontanément, soit suite à un avortement ; seule la boucle infinie ne se termine jamais spontanément. Nous considérons un temps logique découpé en instants successifs, donc si  $t' = t + 1$  par exemple, l'instant  $t'$  suit immédiatement l'instant  $t$ .

Une instruction non-instantanée (qui «prend du temps») permet de suspendre l'exécution d'une composition et de la reprendre à un instant ultérieur, c'est-à-dire de garder active une composition sur plusieurs instants. Cette suspension correspond soit à une pause jusqu'à l'instant suivant (instruction `pause`), soit à l'attente d'un signal donné. L'instruction réactive `await` permet par exemple d'attendre l'arrivée d'un signal, en «gelant» le fil d'exécution (*thread*) dans lequel elle se trouve jusqu'à la prochaine occurrence du signal. Voici un premier exemple de composition en ESTEREL, qui attend l'arrivée des signaux `I1` et `I2` pour émettre le signal `O` (les instants grisés correspondent à la terminaison de la composition) :

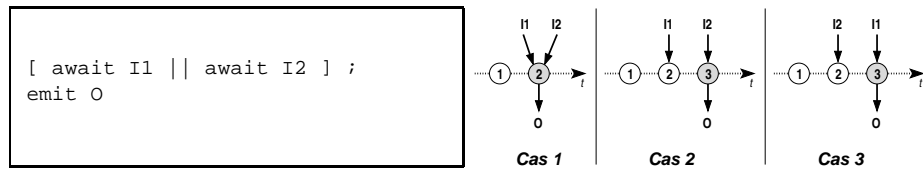


FIG. 5 – Attente concurrente et émission de signaux.

L'opérateur de composition parallèle (`||`) exprime une concurrence explicite déterministe entre deux *threads* d'exécution : il sépare le fil d'exécution courant en deux fils lancés instantanément et s'exécutant en parallèle. Cette instruction se termine à l'instant où les deux composantes concurrentes sont terminées, qu'elles se terminent toutes les deux au même instant ou successivement. Comme le montrent les trois exemples de scénario, la composition parallèle se termine à l'instant où les deux signaux `I1` et `I2` ont été reçus, mais il n'est pas obligatoire que les deux signaux soient reçus simultanément. La séquence (indiquée par l'opérateur `;`) et l'émission de signal sont des instructions instantanées, donc dans l'instant où l'instruction parallèle se termine, le signal `O` est émis et la séquence se termine.

Nous remarquerons que les délimiteurs de bloc `[` et `]` sont nécessaires ici car la séquence est prioritaire sur la composition parallèle. Sans ces crochets, le bloc correspondrait à la composition suivante :

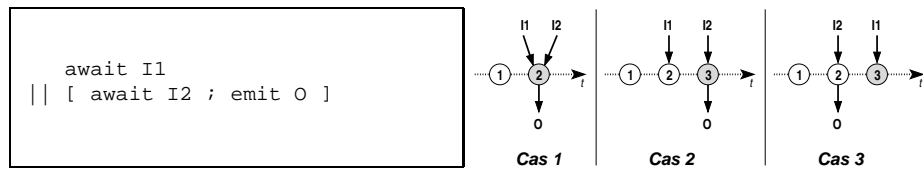


FIG. 6 – De l'importance des délimiteurs de blocs.

Le troisième scénario diffère considérablement de celui de la composition précédente, car le signal `O` est émis quand le signal `I2` est présent, indépendamment du signal `I1`.

### 3.2 Modules

Un programme ESTEREL se présente sous la forme d'un **module**, qui peut être récursivement composé de sous-modules. La communication entre un module et son environnement est réalisée par des signaux. Un module est constitué d'une **interface** et d'un **corps**. L'interface décrit notamment les signaux d'entrée et de sortie du module, c'est-à-dire respectivement les signaux auxquels il doit réagir et les signaux qu'il peut émettre en réponse. Le corps définit le comportement du module, sous la forme d'une composition d'instructions réactives ou impératives.

Lorsqu'il est activé, un module exécute son corps de façon à réagir instantanément à la présence et à l'absence de signaux en entrée et en sortie. Cette exécution aura pour conséquence l'émission de signaux en sortie et/ou la modification de l'état interne du

module. La figure 7 représente un module ESTEREL  $M$  sous la forme d'une boîte noire possédant une interface constituée d'entrées et de sorties pour la réception et l'émission de signaux, et d'une entrée particulière correspondant au déclenchement d'une réaction. Cette dernière entrée sera par la suite considérée comme implicite.

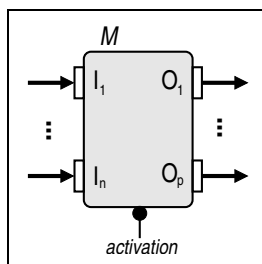


FIG. 7 – Représentation de l'interface d'un module ESTEREL.

Voici un exemple de module ESTEREL, qui émet le signal O1 quand le signal I est présent et le signal O2 sinon, puis attend l'instant suivant. Le test de présence du signal I se fait grâce à l'alternative réactive. L'instruction `pause` ne fait rien et se termine à l'instant suivant :

```

module M :

  % Interface
  input I ;
  output O1, O2 ;

  % Corps
  loop
    present I then
      emit O1
    else
      emit O2
    end present ;
    pause
  end loop

end module

```

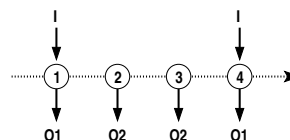


FIG. 8 – Premier module et premières instructions réactives.

L'instruction `pause` est indispensable pour éviter que la boucle infinie s'exécute instantanément, c'est-à-dire qu'elle s'exécute un nombre «infini» de fois au cours du même instant sans rendre la main. Le compilateur ESTEREL doit détecter une telle boucle instantanée et refuser le programme.

L'interface d'un module permet également de déclarer les types, constantes, fonctions et procédures externes utilisés dans le corps du module. La définition d'une donnée déclarée dans l'interface doit être fournie en externe, lors d'une instanciation par un autre module ESTEREL (*cf.* section 3.8) ou dans le langage cible (*cf.* section 4). La section 3.4 présente plus en détail la manipulation des objets déclarés dans l'interface d'un module.

### 3.3 Signaux

Un **signal pur** est un simple indicateur d'événement, tandis qu'un **signal valué** transmet de plus une information typée. Le signal `tick` est un signal pur prédéfini, toujours présent, qui correspond à l'horloge d'activation. Les deux boucles suivantes sont donc équivalentes<sup>5</sup> :

```
loop
  emit 0
each tick
```

```
loop
  emit 0 ;
  pause
end loop
```

Un signal est caractérisé par son statut, c'est-à-dire par sa présence ou son absence au cours d'un instant ; le signal est indéterminé jusqu'à ce que son statut soit connu. Un signal en entrée est présent s'il a été fourni par l'environnement au début de l'instant. Un signal en sortie est considéré comme absent au début d'un instant et devient présent dès qu'une émission explicite est réalisée (`emit`). La diffusion instantanée des signaux implique le partage du statut et de la valeur éventuelle de chaque signal à un instant donné entre toutes les composantes d'un programme. Ce type de communication peut rendre la compréhension de certains programmes difficile, car la présence d'un signal en sortie peut être testée même si le signal vient d'être émis par le programme dans le même instant. Dans l'exemple suivant, le signal `O` sera émis à chaque instant, car le signal `S` est émis à chaque activation du corps :

```
loop
  present S then
    emit 0
  end present
  ||
  emit S
each tick
```

En ESTEREL, le temps est parfois qualifié de «multiforme», car il est caractérisé par des signaux qui ne représentent que rarement le temps physique (par exemple des secondes). L'utilisateur peut considérer que chaque signal définit sa propre unité de temps, la durée de chaque pas de temps pour ce signal pouvant de plus être variable puisque le signal peut être reçu de façon irrégulière.

#### 3.3.1 Expressions de signaux

Il est possible d'utiliser les opérateurs booléens `not`, `and` et `or` dans une expression testant la présence de signaux. L'exemple suivant émet le signal `O1` si les signaux `I1` et `I2` sont présents simultanément ou si le signal `I3` est absent :

```
present [I1 and I2] or [not I3] then
  emit O1
end present
```

Les deux expressions suivantes, utilisant les deux formes abrégées de l'instruction `present`, sont donc équivalentes :

<sup>5</sup>La signification exacte de la construction `loop p each S` est expliquée dans la section 3.6. Considérons pour le moment qu'elle correspond à une boucle qui exécute son corps à chaque réception du signal `S`, donc dans notre exemple à chaque instant.

```
present I else
  emit O
end present
```

```
present not I then
  emit O
end present
```

Il est aussi possible d'indiquer qu'il faut prendre en compte plusieurs réceptions du même signal. Voici un exemple, qui attend trois occurrences du signal I avant de se terminer :

```
await 3 I
```

Les expressions de signaux peuvent être utilisées dans les instructions de préemptions, que nous aborderons en section 3.6.

### 3.3.2 Prise en compte immédiate d'un signal

La plupart des instructions réactives sont «retardées», ce qui signifie que la condition de présence d'un signal n'est valable qu'à partir de l'instant suivant (retard d'un instant) : dans le cas de l'instruction `await` par exemple, si le signal est présent dans l'instant où cette instruction est atteinte, cela n'aura pas d'effet. Le mot-clé `immediate` indique que l'occurrence attendue peut appartenir à l'instant courant, c'est-à-dire qu'un signal doit être pris en compte dès le premier instant d'exécution d'une instruction. La séquence suivante se termine donc instantanément :

```
emit S ;
await immediate S
```

## 3.4 Manipulation de données

Un module ESTEREL peut manipuler des valeurs de type quelconque, à condition que ce type soit prédéfini<sup>6</sup> ou qu'il soit déclaré dans la partie interface du module. Les valeurs manipulées correspondent à des variables, à des constantes ou à l'information transmise par un signal valué (cf. section 3.5). L'alternative classique peut être utilisée pour tester ces valeurs et exécuter des sous-blocs en conséquence.

Il est possible de manipuler des variables locales typées : la déclaration d'une variable définit un bloc à l'intérieur duquel la variable peut être utilisée, éventuellement avec une valeur initiale. L'affectation d'une variable se fait en utilisant l'opérateur `:=`. Il n'y a pas de notion de variable globale au module, contrairement aux constantes, qui sont définies dans l'interface du module et dont la valeur est externe. L'exemple suivant illustre la manipulation d'une constante et de deux variables entières. Le signal `Equal` est émis lorsque les deux variables ont la même valeur :

```
module M :
  output Equal ;
  constant C : integer ;
```

<sup>6</sup>Les types prédéfinis disponibles sont les suivants : `integer`, `float`, `double`, `boolean`, `string`.

```

var X := 0 : integer,
    Y   : integer
in
  Y := C ;
  loop
    if (X = Y) then
      emit Equal ;
      X := 0 ;
      Y := C
    else
      X := X + 1
    end if
  each tick
end var
end module

```

Dans le cas de composantes concurrentes, deux cas de figure seulement sont autorisés, les autres devant être refusés par le compilateur :

- la variable est utilisée en lecture seule dans les différents *threads* ;
- la variable est modifiée dans l'un des *threads*, mais elle n'est ni lue ni modifiée dans les autres.

Ces limitations garantissent à la composition parallèle la propriété remarquable de conserver le déterminisme du système décrit. L'exemple suivant est donc invalide, cette composition n'ayant pas de sens dans un cadre déterministe :

```

  X := X + 1
||
  X := 1

```

Il est possible d'utiliser des fonctions et des procédures externes (par exemple pour manipuler des données de type externe), à condition qu'elles aient été déclarées dans l'interface du module et que leur instantanéité soit garantie. Une fonction est déclarée en précisant le type de ses paramètres et de sa valeur de retour. Pour déclarer une procédure, il faut d'abord donner le type des paramètres en entrée-sortie (modifiables), puis le type des paramètres en entrée ; l'appel de procédure se fait en utilisant l'instruction `call`. L'exemple suivant présente l'utilisation d'une constante de type externe `T` par une fonction et une procédure externes, ainsi que la manipulation de deux variables de type `T` :

```

module M :
  input I ;

  type T ;
  procedure Increment (T) (int) ;
  function Init () : T ;
  function Test (T) : boolean ;

  var X := Init () : T
  in
    loop
      if (Test(X)) then
        call Increment (X) (1)
      end if
    each I
  end var
end module

```

### 3.5 Signaux valués et capteurs

Un signal valué est déclaré comme porteur d'une donnée typée. Il est possible d'initialiser la valeur d'un signal lors de sa déclaration, en utilisant l'opérateur `:=`. La valeur d'un signal est toujours accessible en utilisant l'opérateur `?`, même si le signal est absent. En fait, les instants de présence d'un signal valué correspondent aux instants d'affectation de la valeur du signal : si le signal `S` est absent, `?S` renvoie donc la valeur précédente de `S`. L'exemple suivant, tiré de [14], est une bonne illustration de ce phénomène ; le module `Observateur` émet à chaque instant un signal `O` transportant la valeur courante du signal `I` :

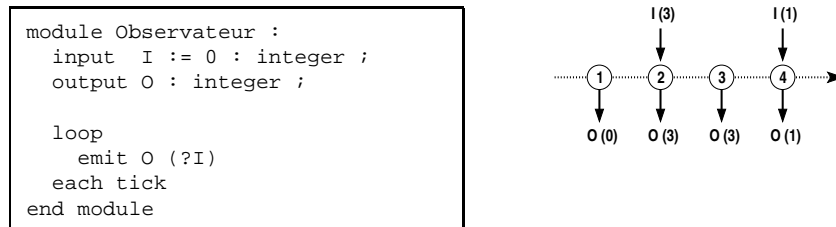


FIG. 9 – Manipulation des valeurs de signaux.

Il est cependant impossible d'utiliser la valeur d'un signal pour calculer une nouvelle valeur pour ce même signal. L'exemple suivant est invalide :

```
emit S (?S + 1)
```

Il ne s'agit en effet pas d'une simple affectation prenant effet après la consultation de la valeur courante. Contrairement à une variable, un signal ne peut pas prendre plusieurs valeurs dans le même instant car, lors de l'émission, la valeur du signal pour l'instant courant est immédiatement modifiée et diffusée : il est évidemment impossible de satisfaire la relation  $S = S + 1$ . Un tel cycle de causalité doit être détecté à la compilation, et le programme doit être refusé.

Un signal valué ne peut donc *a priori* pas être émis plusieurs fois dans le même instant, en particulier dans des composantes concurrentes<sup>7</sup>. En utilisant le mot-clé `combine` lors de la déclaration d'un signal valué en sortie, il est possible de définir une méthode de combinaison, qui sera utilisée dans le cas d'une émission multiple de ce signal. L'exemple suivant, composé de deux *threads*, émet à chaque pas de réaction un signal `O` et incrémente une variable `X`. Dans tous les cas, `O` est émis avec la valeur courante de `X`. Il est émis simultanément avec la valeur 10 lorsque le signal `I` est présent ; grâce à la méthode de combinaison `+`, la valeur du signal `O` sera dans ce cas augmentée de 10 :

<sup>7</sup>L'émission multiple d'un signal pur ne pose pas de problème car le statut du signal n'est pas modifié.

```

module M :
  input I ;
  output O :
    combine integer with + ;

  var X := 0 : integer
  in
    loop
      emit O (X) ;
      X := X + 1
    ||
      present I then
        emit O (10)
      end present
    each tick
  end var
end module

```

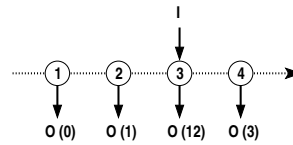


FIG. 10 – Combinaison de signaux de sortie.

Pour le type prédéfini `boolean`, la combinaison peut se faire avec les opérateurs `and` et `or`. Pour les types numériques prédéfinis, les opérateurs `+` et `*` sont utilisables. Dans le cas de types externes, c'est à l'utilisateur de fournir les fonctions externes appropriées, qui doivent être commutatives et associatives.

Un **capteur** est un signal valué dégénéré, qui ne possède pas de statut de présence. Il s'agit en pratique d'une variable externe en lecture seule, dont la valeur peut être récupérée avec l'opérateur `?`. L'événement associé à un capteur n'a pas de sens : tout se passe comme si ce signal était toujours présent. Les capteurs permettent d'acquérir des informations non événementielles en provenance de l'environnement, par exemple la valeur courante d'un thermomètre. Il est donc inutile d'initialiser la valeur d'un capteur. L'exemple suivant réagit au signal `I` en émettant le signal `Fahrenheit` transportant la traduction en degrés Fahrenheit de la valeur du capteur de température `Celsius` :

```

module Thermometre :
  input I ;
  output Fahrenheit : float ;
  sensor Celsius : float ;

  function c2f (float) : float ;

  loop
    emit Fahrenheit (c2f(?Celsius)) ;
  each I
end module

```

### 3.6 Prémption

Un mécanisme de prémption permet d'interrompre un traitement en présence d'événements particuliers, en l'avortant ou en le suspendant. Les instructions de prémption d'ESTEREL permettent d'exprimer de façon concise des comportements sophistiqués ; les opérations disponibles sont la suspension, l'avortement fort et l'avortement faible. Ces constructions, en particulier l'avortement, sont souvent utilisées pour représenter des compositions qui doivent s'exécuter dans un délai précis, écoulé lorsque

le signal de préemption est reçu. Le mécanisme d'exceptions, détaillé dans la section suivante (3.7), permet lui aussi une forme de préemption.

Une activité suspendue peut reprendre au point où elle a été préemptée dès que les conditions de suspension ont disparu. L'instruction `suspend` est retardée, le bloc encapsulé sera donc exécuté au moins une fois, au cours du premier instant :

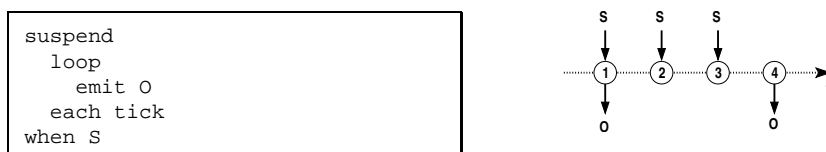


FIG. 11 – Suspension sur réception de signal.

L'instruction d'avortement `abort` permet l'abandon d'un bloc dès réception d'un signal donné. C'est aussi une instruction retardée, donc le signal d'avortement ne sera pas pris en compte lors du premier instant. Si le corps se termine naturellement avant la première occurrence du signal d'avortement, l'ensemble se termine instantanément. Sinon :

- dans le cas de l'avortement fort (`abort`), l'ensemble se termine à l'instant de la réception du signal, sans transmettre le contrôle au corps ;
- dans le cas d'un avortement faible (`weak abort`), le corps s'exécute une dernière fois puis est avorté.

Les deux exemples suivants illustrent l'utilisation des instructions d'avortement fort et faible. Les scénarios montrent la différence entre les deux types d'avortement :

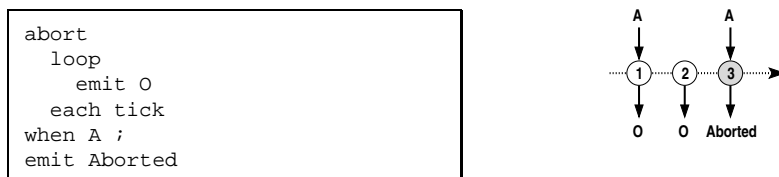


FIG. 12 – Avortement fort sur réception de signal.

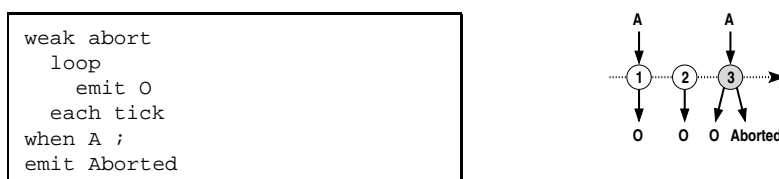


FIG. 13 – Avortement faible sur réception de signal.

En ajoutant le mot-clé `immediate` aux instructions d'avortement fort et faible, nous obtenons deux nouvelles constructions permettant de prendre en compte dès le

premier instant le signal d'avortement. En imbriquant des instructions de préemption, il est possible de définir des hiérarchies de priorités, l'instruction englobante étant prioritaire. Il est aussi possible d'ajouter une clause de traitement du cas où le signal d'avortement a été reçu.

Il existe par ailleurs deux formes de boucles étendues qui utilisent l'avortement de façon masquée. L'instruction `loop p each S` a le comportement suivant :

- si *p* termine, il faut attendre le signal *S* pour le relancer. Cela correspond au comportement intuitif que nous avons donné en section 3.3 ;
- si le signal *S* est reçu avant que *p* ne se termine, alors *p* est avorté et relancé instantanément.

La boucle `every` a presque le même comportement, mais elle attend une première occurrence du signal *S* pour lancer son corps.

### 3.7 Trappes

Le mécanisme de trappes d'ESTEREL permet de s'échapper d'un bloc d'instructions en précisant le motif de sortie ; une trappe déclenchée dans un bloc peut être rattrapée si un traitement spécifique est nécessaire. L'instruction `trap` définit une trappe qui peut être utilisée dans son corps : si ce corps se termine, naturellement ou en déclenchant la trappe (instruction `exit`), l'ensemble se termine instantanément. Dans l'exemple suivant, le corps du bloc `trap` déclenche la trappe *T* lorsque le signal *I* est présent, et émet le signal *O* sinon. Le bloc `trap` se termine ici seulement si la trappe est déclenchée ; le signal *E* est alors émis instantanément :

```

trap T in
  loop
    present I then
      exit T
    end present ;
    emit O
  each tick
end trap ;
emit E

```

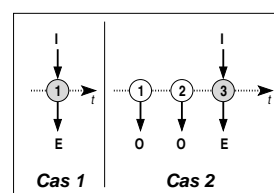


FIG. 14 – Utilisation d'une trappe.

Le déclenchement d'une trappe correspond à un avortement faible : si le corps du bloc `trap` contient des branches parallèles, dont l'une a déclenché la trappe, les autres s'exécutent quand même une dernière fois. La branche ayant déclenché la trappe se termine bien sûr immédiatement. Si un traitement est prévu suite à une sortie de trappe, il s'exécutera après que les branches concurrentes se soient terminées, dans le même instant. Le mot-clé `handle` permet de spécifier un traitement à effectuer lorsqu'une trappe est déclenchée. Il est aussi possible d'utiliser des trappes valuées, dont la valeur de retour peut être récupérée avec l'opérateur `??`. Si plusieurs blocs `trap` sont imbriqués, le bloc `trap` englobant, qui déclenche une trappe simultanément aux blocs encapsulés, est prioritaire (comme pour la préemption).

### 3.8 Composition de modules

Un module ESTEREL peut instancier d'autres modules en utilisant l'instruction `run` et en spécifiant au besoin les correspondances (ou substitutions) entre les signaux. L'exemple suivant montre comment créer, dans un module `P`, deux instances `M1` et `M2` d'un module `M`, et comment faire correspondre les signaux `I1` et `I2` de `P` aux signaux `I` de `M1` et `M2`. Il est inutile de préciser la substitution de deux signaux portant le même nom ; le signal `O` de `P` correspondra donc au signal `O` de `M1` et de `M2` :

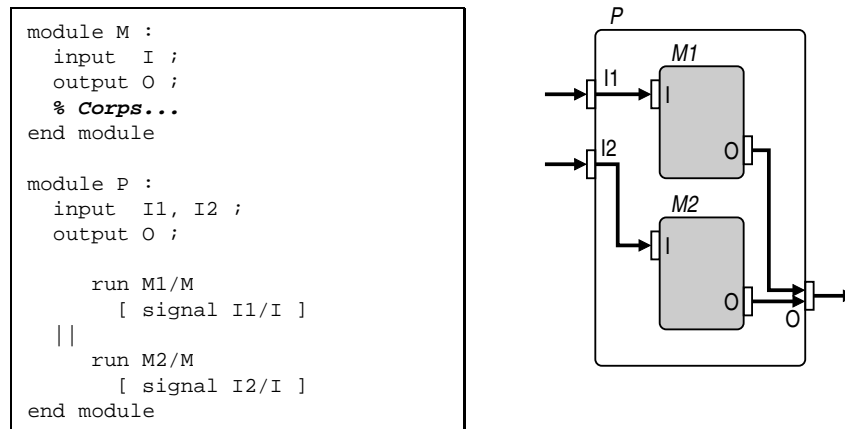


FIG. 15 – Exemple de composition de modules avec substitution de signaux.

Nous pouvons alors considérer une hiérarchie entre les modules : un module qui en instancie un autre peut être appelé module-père et un module instancié est un module-fils. L'instanciation d'un module correspond à la recopie exacte du corps de ce module, et à l'ajout des déclarations de son interface à celles de l'interface du module-père. La substitution ne se limite pas à la correspondance entre signaux : elle s'applique à tout ce qui peut être déclaré dans une interface de module, c'est-à-dire aux types, aux constantes, aux fonctions et aux procédures. Cela permet en particulier de définir des modules génériques, paramétrés lors de l'instanciation.

Les signaux locaux permettent la communication entre modules exécutés au sein d'un même module englobant. Plusieurs signaux locaux peuvent être définis dans le même bloc ; il est également possible de déclarer à cet endroit une fonction de combinaison de signaux (*cf.* section 3.5). Dans l'exemple suivant, le module `Q` instancie le module `M` en `M1` et `M2`, et crée un signal local `L` permettant de faire correspondre le signal de sortie `O` de `M1` et le signal d'entrée `I` de `M2` :

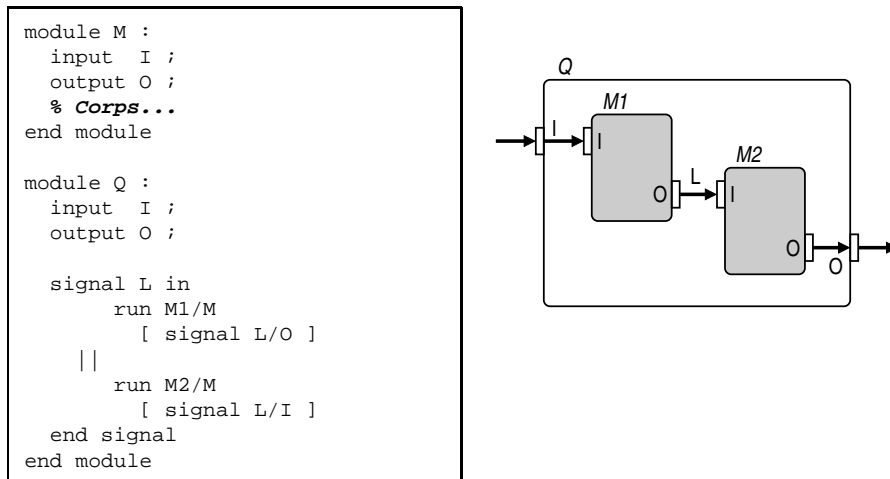


FIG. 16 – Exemple de composition de modules utilisant un signal local.

La diffusion instantanée de signaux entre modules peut être bidirectionnelle, à condition qu'elle n'introduise pas de cycle de causalité. Cette propriété impose une limitation importante car les cycles sont détectés uniquement à la compilation : la création et la destruction dynamiques de modules sont impossibles. L'architecture d'un système réactif décrit en ESTEREL ne peut donc pas évoluer au cours de son exécution.

### 3.9 Tâches asynchrones

Tous les traitements qui ont lieu au cours d'une réaction d'un programme ESTEREL sont considérés comme instantanés, en particulier les appels de fonctions et de procédures externes. Il est cependant parfois nécessaire de faire appel à des traitements dont la durée ne peut pas être négligée, en particulier lorsqu'il s'agit de traitements transformationnels dont il faut attendre le résultat. Une tâche asynchrone permet de lancer, parallèlement à un programme ESTEREL, une opération dont la durée dépasse les bornes fixées pour garantir la réactivité du système, comme un calcul long ou le déplacement d'un bras de robot. Une tâche est lancée avec l'instruction `exec`, puis gérée de façon logique avec les instructions que nous avons introduites précédemment, en particulier à l'aide des instructions de préemption.

Les informations utiles pour manipuler une tâche sont ses instants de début et de fin (spontanée ou suite à un avortement). Il faut que le programme se synchronise avec l'environnement, chargé d'exécuter effectivement la tâche : il faut demander à l'environnement de lancer la tâche, puis attendre qu'elle se termine et éventuellement récupérer son résultat. Il faut également prévenir l'environnement qu'une tâche doit être suspendue ou avortée. La déclaration et le lancement d'une tâche sont syntaxiquement similaires à la déclaration et à l'appel d'une procédure.

Le *thread* qui lance la tâche est bloqué en attente de la terminaison de cette tâche. S'il y a d'autres *threads*, ils s'exécutent en parallèle de la tâche ; le programme n'est alors pas suspendu en attendant la fin de la tâche. Quand la tâche est terminée, les références passées en paramètres à la tâche sont mis à jour instantanément et l'instruction `exec` se termine. Les signaux spécifiques aux événements de fin de tâche sont déclarés

à l'aide du mot-clé `return`. Le signal de retour ne peut pas être émis par le programme lui-même, il est émis par l'environnement à l'instant où la tâche se termine ; ce signal pourra alors être testé afin de savoir si la tâche s'est terminée spontanément dans l'instant courant ou si elle a été avortée. L'exemple suivant décrit une tâche chargée de calculer une trajectoire à partir de coordonnées :

```

module M :
  type Coordinates, Trajectory ;

  input  Current : Coordinates ;
  output NewTrajectory : Trajectory ;
  return R ;

  task ComputeTrajectory (Trajectory) (Coordinates) ;

  var T : Trajectory in
    [ loop
      await Current ;
      exec ComputeTrajectory (T) (?Current) return R ;
      emit NewTrajectory (T)
    end loop ]
  ||
  p % Corps à exécuter en parallèle
end var
end module

```

Une tâche peut être préemptée, c'est-à-dire avortée ou suspendue. Dans l'exemple suivant, la tâche est avortée et instantanément relancée lorsque le signal `I` est reçu, ce qui signifie que l'environnement est prévenu qu'il doit avorter et relancer cette tâche :

```

loop
  exec T (X) () return R
each I

```

Les tâches asynchrones sont entre autres utilisées pour implémenter un mécanisme de rendez-vous entre modules réactifs, dans le cadre des CRP (*Communicating Reactive Processes*) : l'envoi de messages (bloquant ou non) et la réception (rendez-vous) sont réalisés par des tâches spécifiques [5].

### 3.10 Compilation d'un programme ESTEREL

Le compilateur ESTEREL produit entre autres un fichier `C`, représentant un automate à états finis ou un circuit booléen. Ce fichier contient en particulier une fonction d'activation et une fonction de positionnement pour chaque signal d'entrée. L'utilisateur doit fournir une fonction de positionnement pour chaque signal de sortie, dans un fichier `C` séparé. L'exécution d'une réaction est obtenue en appelant d'abord la fonction de positionnement de chaque signal d'entrée présent, puis la fonction d'activation. L'émission d'un signal en sortie provoque l'appel de la fonction correspondante, définie par l'utilisateur. L'exemple de la figure 17, inspiré d'un exemple de [14], montre la correspondance entre un module ESTEREL (fichier `M.str1`) et les interfaces des fonctions `C` utilisées, présentes dans les fichiers `M.c` (généré par le compilateur) et `M_outputs.c` (fourni par l'utilisateur) :

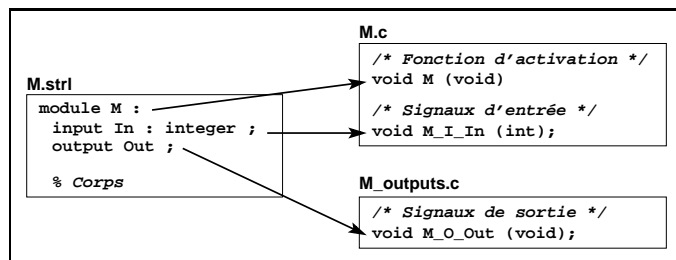


FIG. 17 – Exemple de correspondance ESTEREL/C.

## 4 Machine d'exécution

L'approche synchrone facilite considérablement la spécification de systèmes réactifs, mais un système décrit dans un langage synchrone comme ESTEREL n'est pas directement exécutable. En particulier, la notion idéale de réaction s'exécutant en temps nul n'est évidemment pas réalisable car il n'existe pas de machine infiniment rapide. Nous nous intéressons plus particulièrement aux caractéristiques propres à ESTEREL, il faut donc de plus pouvoir traduire, pour le module synchrone, les événements asynchrones en provenance du monde réel en signaux logiques pour lesquels la notion de simultanéité a un sens. La mise en œuvre d'un système synchrone nécessite l'utilisation de mécanismes d'interfaçage entre ce système et son environnement ; ces mécanismes sont regroupés dans ce qui est communément appelé une **machine d'exécution**. Une machine d'exécution pour ESTEREL est donc chargée de réaliser une approximation acceptable de l'hypothèse synchrone, de faire la correspondance entre les événements externes et les signaux du module définissant le système synchrone, et de déclencher les réactions de ce module. Elle permet également de gérer les tâches asynchrones. La figure 18 schématise le fonctionnement d'une machine d'exécution pour module ESTEREL.

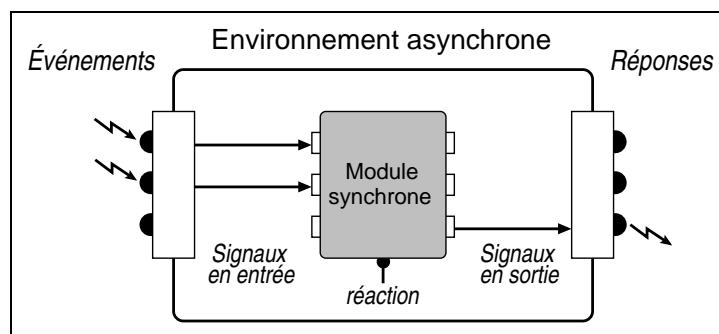


FIG. 18 – Principes d'une machine d'exécution pour ESTEREL.

Le modèle synchrone définit des réactions de durée nulle, ce qui est en pratique irréalisable. Cette propriété est donc approximée en assurant l'**atomicité** de chaque réaction. Il faut pour cela poser deux contraintes fortes sur les fréquences d'activation du module et d'arrivée des signaux :

1. l'exécution de la réaction doit être complète, il ne faut donc pas activer à nouveau

le module s'il est en cours de réaction ;

2. les signaux en entrée doivent être positionnés avant l'activation du module et ne doivent pas être modifiés pendant la réaction. La perception que le module a de son environnement doit en effet être figée pendant toute la durée de la réaction.

La machine d'exécution peut déclencher la réaction d'un module selon deux stratégies au choix :

- l'activation se fait sur l'arrivée de signaux. Cette stratégie «par interruption» est particulièrement adaptée aux systèmes à événements sporadiques, qui sont souvent «au repos» ;
- les réactions sont déclenchées régulièrement, indépendamment des occurrences de signaux. Cette stratégie d'échantillonnage périodique s'applique plutôt aux systèmes qui doivent réagir «de façon continue» à leur environnement.

Pour que le système défini soit réactif vis-à-vis de son environnement, il faut garantir que l'exécution des réactions sera assez rapide. La vérification de cette propriété se fait *a posteriori*, en exécutant le système.

La machine d'exécution et le système synchrone s'exécutent en parallèle, car la machine d'exécution doit prendre en compte les événements en provenance de l'environnement asynchrone pendant que les modules synchrones exécutent un pas de réaction. La prise en compte des événements asynchrones est habituellement réalisée en les mémorisant jusqu'au prochain instant, puis en les traduisant en signaux d'entrée présents avant de déclencher la réaction suivante. Un module activé réagit ainsi à tous les événements observés depuis le début de la réaction précédente. Cette mémorisation, à la charge de la machine d'exécution, garantit que les signaux d'entrée ne seront pas modifiés pendant l'exécution d'une réaction<sup>8</sup>.

La machine d'exécution sert par ailleurs d'interface entre les modules et le langage cible implémentant les sous-programmes synchrones et les tâches asynchrones. En particulier, les modules ESTEREL n'ont qu'une vision logique des tâches asynchrones : leur implémentation est laissée au soin de l'utilisateur. La machine d'exécution permet d'exécuter ces tâches et de les gérer, par exemple au moyen d'un exécutif multi-tâches.

La mise en œuvre effective d'un système synchrone est souvent considérée comme un détail d'implémentation, traité au cas par cas. Par exemple, une machine d'exécution pour ESTEREL utilisant les services du système d'exploitation multi-tâches temps-réel CHORUS a été développée par le CNET<sup>9</sup> pour décrire le comportement d'objets multimédia et gérer la synchronisation de ces objets [17]. H. Boufaïed propose pour sa part une architecture générique de machine d'exécution, indépendante de la cible d'exécution [6]. Son modèle permet de définir différentes stratégies d'acquisition et d'émission de signaux, d'enchaînement des phases de réaction et de traitement des anomalies observées. Il permet de décrire la plupart des mécanismes d'exécution de processus synchrones et il intègre le traitement de tâches asynchrones. Les contrôleurs d'entrée-sortie réalisent l'interface entre le monde extérieur et le processus synchrone.

<sup>8</sup>H. Boufaïed a montré que la mémorisation de l'historique des événements permet en particulier de conserver une approximation satisfaisante malgré les réceptions multiples de signaux possibles dans le cas d'un système temps-réel à contraintes temporelles relatives [6].

<sup>9</sup>Le CNET (Centre National d'Études des Télécommunications) est maintenant appelé FT R&D (France Télécom, Recherche et Développement).

## 5 Réseaux de processus réactifs

Le modèle de **réseaux de processus séquentiels** a été défini afin d'étendre la sémantique dénotationnelle. Selon F. Boussinot, cette solution est modulaire et élégante mais elle manque d'expressivité ; il l'a donc étendue en proposant le modèle de **réseaux de processus réactifs**. Ces deux modèles sont entre autres décrits en détail dans son ouvrage consacré à la programmation réactive [11]. Dans les deux cas, le comportement d'un réseau de processus est parfaitement déterministe.

Un processus séquentiel est un programme possédant une mémoire et des ports de communication. Il peut recevoir des messages sur ses ports d'entrée et envoyer des messages par l'intermédiaire de ses ports de sortie. Un réseau de processus séquentiels est formé en connectant les ports des processus par des canaux. Un canal est une file FIFO<sup>10</sup> de taille infinie. Un processus communique de façon asynchrone avec les autres processus tout au long de son exécution, en déposant des messages sur les canaux connectés à ses ports de sortie et en récupérant les messages dans les canaux connectés à ses entrées. Cette récupération est bloquante : un processus qui tente de prendre un message dans un canal vide est bloqué jusqu'à ce qu'un message soit déposé sur le canal, auquel cas il peut reprendre son exécution. L'envoi et la réception de messages sont les seuls mécanismes de synchronisation disponibles ; il n'y a en particulier aucun partage de variables globales.

Un canal a au plus un processus producteur, capable de déposer des messages, et un processus consommateur, capable de récupérer ces messages. Les canaux sans producteurs constituent les entrées du réseau de processus et les canaux sans consommateurs correspondent aux sorties du réseau ; ces entrées et ces sorties permettent l'interfaçage avec l'environnement. La structure d'un réseau peut être modifiée dynamiquement, au cours de l'exécution des processus : le nombre de processus et de canaux peut évoluer, et les interconnexions peuvent être modifiées. Le déterminisme d'un réseau de processus séquentiels est garanti si les hypothèses suivantes sont respectées :

- un processus ne peut pas tester si un canal est vide, afin que le comportement du réseau ne dépende pas des différentes vitesses d'exécution des processus (ce qui le rendrait indéterministe) ;
- un processus ne peut pas tenter de récupérer un message sur plusieurs canaux à la fois.

Ces deux contraintes limitent l'expressivité du modèle. Il n'est par exemple pas possible de gérer des canaux avec des priorités différentes, car il n'est possible ni d'attendre sur plusieurs canaux, ni de tester que les canaux les plus prioritaires contiennent effectivement des messages. En ajoutant la notion d'instant au modèle précédent, il devient possible de tester si un canal est vide, sans pour autant perdre le déterminisme du réseau de processus. Grâce à l'introduction de cette notion, le modèle de réseaux de processus réactifs est plus expressif tout en préservant le déterminisme. Tous les processus partagent la même notion d'instant et sont exécutés à chaque instant. Un processus a fini de réagir lorsqu'il se termine, lorsqu'il est explicitement suspendu jusqu'à l'instant suivant ou lorsqu'il est bloqué en attente d'un message sur un canal vide. Un instant du réseau se termine lorsque tous les processus ont fini de réagir, c'est-à-dire quand le système s'est stabilisé. Le problème des différentes vitesses d'exécution des processus disparaît, puisque tous les processus s'exécutent maintenant au même

---

<sup>10</sup>FIFO : *First In, First Out* (premier entré, premier sorti).

rythme.

Pour éviter les problèmes de cohérence vis-à-vis de la primitive de test d'un canal vide, ce test est systématiquement différé au début de l'instant suivant. Il est en effet plus simple de garantir qu'un message n'a pas été reçu pendant l'instant courant si ce test est réalisé à la fin de l'instant ; la réaction à l'absence de message peut alors se faire au tout début de l'instant suivant. La primitive de test d'un canal vide fonctionne comme suit :

- si un message a été reçu, alors la primitive retourne «vrai» et termine dans l'instant ;
- sinon, l'exécution est bloquée jusqu'à l'instant suivant et la primitive retournera «faux» seulement à cet instant.

## 6 Objets synchrones et objets réactifs

Les objets synchrones [7, 9, 8] et les objets réactifs [13, 11] sont des modèles très différents visant tous deux à concilier l'approche synchrone et l'approche objet. L'intégration de modules synchrones dans un langage orienté-objets permet de faire coopérer des programmes synchrones et non-synchrones au sein d'une même application, et de profiter à la fois de la rigueur du modèle synchrone et de la souplesse d'un langage orienté-objets. En particulier, il devient possible de créer et de détruire dynamiquement des objets correspondant aux modules synchrones, alors que cette dynamique manque cruellement aux langages synchrones classiques. En contrepartie, il n'est pas possible d'utiliser les outils de vérification formelle des langages synchrones pour valider statiquement le comportement global d'un ensemble d'objets synchrones ou réactifs.

Les deux modèles décomposent les applications en zones réactives, dont les objets partagent la même notion d'instant. Ces zones utilisent des mécanismes synchrones pour la communication interne et des mécanismes asynchrones pour communiquer entre elles, comme l'illustre l'exemple de la figure 19.

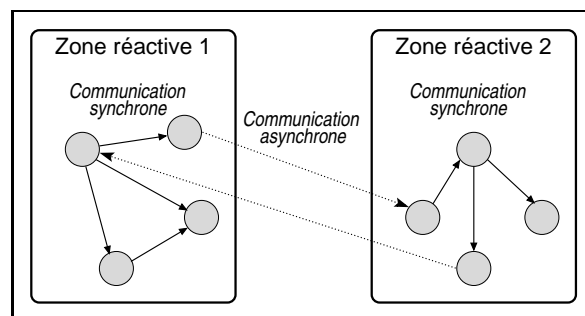


FIG. 19 – Communication entre objets synchrones ou réactifs.

Les **objets synchrones**, définis par F. Boulanger, sont des objets qui encapsulent des modules synchrones compilés ; ils sont explicitement interconnectés pour former des réseaux d'objets synchrones, qui partagent la même notion d'instant global. Les **objets réactifs**, définis par F. Boussinot, sont pour leur part constitués de méthodes

réactives correspondant à des modules synchrones ; ces méthodes s'exécutent en parallèle et leur comportement est décrit en REACTIVE-C, langage basé sur le langage C et développé par F. Boussinot [12, 11] ; les comportements des programmes sont définis en termes de réactions à des activations. REACTIVE-C est considéré comme un langage d'assemblage permettant d'implémenter des formalismes de plus haut niveau basés sur la même notion d'instant. Nous nous intéressons plus particulièrement au modèle d'objets synchrones, c'est pourquoi nous décrivons plus succinctement le modèle d'objets réactifs.

## 6.1 Objets synchrones

Un objet synchrone est la traduction dans un langage à objets d'un module synchrone. Son comportement est produit par le compilateur de langage synchrone à partir du code du module, pour être ensuite encapsulé dans un objet doté d'une unique méthode d'activation et d'une méthode d'accès pour chaque signal d'entrée et de sortie. Les mécanismes de communication entre objets synchrones doivent respecter la sémantique synchrone. Le comportement d'un réseau d'objets synchrones interconnectés est identique à celui d'un programme synchrone constitué des modules synchrones correspondants ; un tel réseau peut donc être considéré comme un système synchrone à interfacer avec l'environnement asynchrone, lui-même composé des autres objets du langage. F. Boulanger propose de plus un ensemble d'outils pour la définition d'objets synchrones, en particulier le langage MDL permettant la composition de modules synchrones.

### 6.1.1 Communication et séquençement synchrone

Les signaux d'entrée et de sortie sont des attributs de l'objet synchrone. Tout signal d'entrée possède une méthode de connexion prenant comme argument un signal de sortie ; cette méthode permet de déclarer explicitement des connexions point-à-point unidirectionnelles entre objets synchrones. Des mécanismes de vérification assurent qu'un signal d'entrée est bien connecté à un signal de sortie du même type.

Les objets synchrones d'un même réseau ont une notion d'instant commune ; cela signifie que les valeurs des signaux échangés ne sont pas modifiées au cours d'un instant. Avant de pouvoir utiliser les signaux de sortie d'un objet synchrone, il faut que ces signaux aient été positionnés ; il faut donc que l'objet en question ait réagi. Si le graphe de dépendances entre objets synchrones interconnectés est acyclique, il est possible de déterminer un ordre d'activation des objets synchrones. Cette opération, appelée **séquençement synchrone**, est confiée à un ordonnanceur, qui doit ensuite faire réagir les objets synchrones dans l'ordre choisi. Le graphe peut contenir des cycles à condition qu'ils ne soient pas instantanés, c'est-à-dire qu'ils contiennent des «retards». Un **retard** est un objet particulier, qui conserve le signal reçu à l'instant courant pour le délivrer à l'instant suivant ; il possède un seul signal d'entrée et un seul signal de sortie.

Les dépendances entre objets synchrones peuvent être modifiées au cours d'un instant, par l'ajout, la destruction, la connexion et la déconnexion des objets. Les requêtes de modification de la topologie du graphe de dépendances sont mémorisées dans l'instant courant, afin de reconstruire le graphe et déterminer un nouvel ordre d'activation à la fin de la réaction. Une nouvelle connexion peut être refusée si elle engendre un cycle de dépendance.

### 6.1.2 Communication asynchrone

La communication entre les objets synchrones et les autres objets du langage se fait au travers d'une interface permettant de traduire les signaux synchrones en événements asynchrones et *vice-versa*. Les objets constituant cette interface, appelés **objets d'interface**, peuvent servir d'échantillonneurs ou peuvent déclencher une réaction sur l'arrivée de certains événements asynchrones, selon la stratégie choisie.

### 6.1.3 Définition et composition de modules

À partir de la description en OC<sup>11</sup> d'un module synchrone, le compilateur OCC++ génère soit une description MDL du module, soit une classe C++ synchrone encapsulant le comportement du module. MDL est un langage de description de modules synchrones par composition et héritage de modules existants. L'héritage de modules permet principalement de ne pas avoir à redéfinir les signaux d'entrée/sortie. Le compilateur MDLC produit une classe C++ synchrone à partir de la description MDL, en détectant les cycles de causalité. La machine d'exécution d'un réseau d'objets synchrones se présente sous la forme d'une bibliothèque de classes (`libSync`) qui fournit l'environnement nécessaire aux classes synchrones produites par OCC++ ou MDLC.

Un module composite est construit à partir d'autres modules connectés de manière statique. Voici un exemple, tiré de [9], qui définit en MDL un module à partir de deux sous-modules interconnectés :

```
compo : {
  input: integer e; // Signaux du module compo
  output: s;

  A mod1, mod2;    // Sous-modules utilisés
  mod1.i = e;      // e est l'entrée i de mod1
  s = mod2.o;      // s est la sortie o de mod2
  mod2.i << mod1.o; // L'entrée i de mod2 est connectée
                  // à la sortie o de mod1.
}
```

## 6.2 Objets réactifs

Dans le modèle ROM (*Reactive Object Model*), une application est constituée d'objets qui exécutent en parallèle des comportements réactifs et communiquent par des appels de méthodes traités instantanément. Un objet réactif encapsule des données, partagées par ses méthodes. Le corps d'une méthode est décrit en REACTIVE-C. Il existe un objet dit initial, possédant une unique méthode chargée de contrôler le déroulement des cycles du système d'objets. Les objets peuvent être clonés et les méthodes peuvent être ajoutées dynamiquement.

Une méthode correspond à un module ESTEREL : elle ne peut être activée qu'une seule fois dans un même instant et elle peut être suspendue jusqu'à un instant ultérieur. Un objet correspond donc à un système composé de modules synchrones. L'exécution d'un objet pour un instant se termine quand toutes les méthodes sont terminées ou suspendues, c'est-à-dire quand le système de modules synchrones est devenu stable. Un

<sup>11</sup>OC (*object code*) est une syntaxe compilée commune à ESTEREL et LUSTRE.

instant du système d'objets réactifs se termine lorsque tous les objets ont terminé leur réaction. Dans une même zone réactive, les méthodes sont appelées de façon asynchrone. Ces appels sont traités instantanément et les méthodes s'exécutent en parallèle. Les paramètres d'appel de méthode peuvent être de n'importe quel type, éventuellement des objets ou d'autres méthodes. À chaque cycle d'activation, les méthodes en attente sont exécutées et les appels asynchrones instantanés sont réalisés.

Comme le modèle de réseaux de processus réactifs, la programmation par objets réactifs introduit une contrainte supplémentaire par rapport à ESTEREL : pour éviter les cycles de causalité liés à la réaction à l'absence de signaux, il est interdit de réagir instantanément à l'absence d'un événement. Cette réaction est reportée nécessairement à l'instant suivant.

## 7 Synthèse

D. Harel et A. Pnueli ont donné des systèmes réactifs l'image de boîtes noires qui réagissent de manière réflexe aux stimuli de l'environnement en changeant d'état et en produisant à leur tour des stimuli. Le modèle synchrone facilite considérablement la description de ce type de systèmes, en considérant des éléments s'exécutant en parallèle et partageant la même notion d'instant ; ces éléments s'exécutent naturellement au même rythme, sans avoir besoin de mettre en place un ordonnanceur de processus concurrents.

ESTEREL est un langage synchrone de nature impérative, adapté à la description de systèmes temps-réel critiques ; il est par exemple utilisé chez DASSAULT-AVIATION pour concevoir des systèmes avioniques. Ce langage permet de définir un système réactif sous la forme d'un module, constitué de composantes parallèles communiquant par diffusion instantanée de signaux. La description du comportement d'un module est réalisée à l'aide d'instructions réactives explicites, comme la pause ou l'attente retardée d'un signal. ESTEREL offre une remarquable puissance d'expression, tout en garantissant le déterminisme du système décrit. En pratique, ce langage nécessite la mise en œuvre d'une machine d'exécution, réalisant l'interface entre le système synchrone et son environnement asynchrone. Des outils de vérification formelle sont disponibles, permettant de concevoir et de valider des comportements complexes s'exécutant en parallèle. L'architecture d'un système réactif décrit en ESTEREL est statique, ce qui permet de garantir la validité du système pendant toute la durée de son exécution ; par contre, cette caractéristique ne correspond pas à nos besoins.

Le modèle d'objets synchrones fournit une solution plus dynamique de description de systèmes réactifs. Des objets encapsulant des modules synchrones sont interconnectés pour former des réseaux d'objets synchrones. Plusieurs objets synchrones peuvent être connectés à une même entrée d'un autre objet synchrone. Les objets et les connexions peuvent être ajoutés et retirés du réseau à tout moment de l'exécution du système, et les connexions peuvent être modifiées. Le séquençage synchrone assure le respect du modèle synchrone et les boucles de causalité sont évitées par l'introduction d'objets retard. La communication asynchrone est possible avec les objets non synchrones du système, constituant l'environnement du réseau d'objets synchrones ; cette communication se fait par l'intermédiaire d'objets d'interface. Les réseaux de processus réactifs constituent également une solution dynamique et déterministe, connectant

par des canaux FIFO des processus s'exécutant en parallèle. Un processus peut tester qu'un canal est vide pour l'instant courant, ce qui correspond au test d'absence de signal en ESTEREL.

Bien qu'ils ne fournissent pas explicitement de mécanisme d'arbitrage, les modèles de réseaux d'objets synchrones et de processus réactifs ressemblent étrangement aux architectures de sélection de l'action basées sur les comportements. Ces modèles, ainsi que le langage ESTEREL, permettent de plus de décrire des systèmes qui doivent réagir à un environnement asynchrone de façon déterministe ; la communication entre les éléments d'un tel système est synchrone car la présence des signaux ou des messages est bornée par la notion d'instant. Il semble que l'association des travaux concernant d'une part les architectures d'agents distribuées et d'autre part les systèmes réactifs synchrones pourrait se révéler fructueuse. C'est pourquoi nous nous avons choisi de définir notre propre architecture d'agents autonomes, basée sur le modèle synchrone ; nous nous sommes ensuite inspirés de la syntaxe d'ESTEREL pour définir la partie réactive du langage MARVIN [19].

## Références

- [1] ANDRÉ C. *Representation and analysis of reactive behaviors : a synchronous approach.* in (CESA'96), pp. 19–29. IEEE-SMC, 1996. Lille (France).
- [2] BERRY G. *The foundations of Esterel.* in G. Plotkin, C. Stirling et M. Tofte (éds), *Proof, language and interaction : essays in honour of Robin Milner.* MIT Press, 1998.
- [3] BERRY G. *The Esterel v5 language primer.* Manuel de référence, Centre de Mathématiques Appliquées (INRIA et École des Mines de Paris), 1999.
- [4] BERRY G., COURONNÉ P. et GONTHIER G. *Programmation synchrone des systèmes réactifs : le langage Esterel.* in *Technique et Science Informatiques (TSI)*, Vol. 6(4), pp. 305–316. Hermès, 1987.
- [5] BERRY G., RAMESH S. et SHYAMSUNDER R.K. *Communicating reactive processes.* in *Twentieth ACM Symposium on Principles of Programming Languages (POPL'93)*, pp. 85–98. 1993. Charleston (USA).
- [6] BOUFAÏED H. *Machines d'exécution pour langages synchrones.* Thèse de doctorat, Université de Nice-Sophia Antipolis, novembre 1998.
- [7] BOULANGER F. *Intégration de modules synchrones dans la programmation par objets.* Thèse de doctorat, Université de Paris XI, Orsay, décembre 1993.
- [8] BOULANGER F., ANDRÉ C., PÉRALDI M.A., RIGAULT J.P. et VIDAL-NAQUET G. *Objets et programmation synchrone.* in *Modélisation des Systèmes Réactifs*, pp. 55–62. 1996. Brest (France).
- [9] BOULANGER F., DELEBECQUE H. et VIDAL-NAQUET G. *Intégration de modules synchrones dans un langage à objets.* in *Real-Time Systems Conference*, pp. 245–260. 1994. Paris (France).
- [10] BOURNAI P., CHÉRON B., GAUTIER T., HOUSSAIS B. et LE GUERNIC P. *SIGNAL manual.* Manuel de référence, IRISA (Rennes), juillet 1993.
- [11] BOUSSINOT F. *La programmation réactive : application aux systèmes communicants.* Collection technique et scientifique des télécommunications (CNET/ENST). Masson, 1996.

- [12] BOUSSINOT F. et DOUMENC G. *Manuel de référence de RC*. Manuel de référence, Centre de Mathématiques Appliquées (École des Mines de Paris), 1992.
- [13] BOUSSINOT F., DOUMENC G. et STEFANI J.B. *Reactive objects*. Rapport de recherche 2664, INRIA, 1995.
- [14] HAINQUE O. *Compilation séparée et exécution distribuée d'applications synchrones modulaires programmées en Esterel*. Thèse de doctorat, École Nationale Supérieure des Télécommunications, juin 2000.
- [15] HALBWACHS N., CASPI P., RAYMOND P. et PILAUD D. *The synchronous data-flow programming language LUSTRE*. in *Special issue on Another Look at Real-time Systems, Proceedings of the IEEE*, Vol. 79, pp. 1305–1320. 1991.
- [16] HAREL D. et PNUELI A. *On the development of reactive systems*. in K.R. Apt (éd.), *Logics and Models of Concurrent Systems (NATO ASI Series)*, Vol. 13, pp. 477–498. Springer-Verlag, 1985.
- [17] HORN F. et STEFANI J.B. *On programming and supporting multimedia object synchronization*. *The Computer Journal*, Vol. 36(1) (1993) 4–18.
- [18] MOREAU G. *Modélisation du comportement pour la simulation interactive : application au trafic routier multimodal*. Thèse de doctorat, IFSIC / IRISA, novembre 1998.
- [19] RICHARD N. *Description de comportements d'agents autonomes évoluant dans des mondes virtuels*. Thèse de doctorat, ENST, octobre 2001.