Programmation synchrone : Propriétés dans une réaction

Charles André* — **Robert de Simone****

* *Laboratoire I3S, UMR 6070, Université de Nice/CNRS*
BP121 – 06903, Sophia Antipolis cédex, France
{andre@unice.fr}

** *INRIA Sophia Antipolis*
BP 93 – 06902 Sophia Antipolis cédex, France
{rs@sophia.inria.fr}

RÉSUMÉ. *Les formalismes impératifs synchrones (Esterel et SyncCharts) permettent d'exprimer des comportements réactifs complexes. La connaissance de la sémantique de ces modèles est indispensable si on veut exploiter pleinement leur richesse expressive et éviter les "cycles de causalité". La première partie de ce papier rappelle les grandes lignes de la sémantique constructive. Cette sémantique est utilisée dans la seconde partie pour analyser des propriétés dans l'instant portant sur les ordres partiels d'exécutions d'actions simultanées. Il est montré comment les fonctionnalités du compilateur, issues de la sémantique constructive, permettent une telle analyse. La technique est illustrée par un exemple de zFIFO (0 fall-through time FIFO queue) un système qui présente de nombreuses actions simultanées dont l'ordonnancement est critique.*

ABSTRACT. *Complex reactive behaviors can be expressed by synchronous imperative formalisms like Esterel or SyncCharts. To make the best of these models and to avoid the pitfall of "causality cycles", the user has to understand the underlying semantics, known as the "constructive semantics". The first part of this paper is an informal introduction to this semantics. In the second part, this semantics is used to analyze "intra-instant properties" (partial ordering of simultaneous action executions). It appears that the compiler, which implements the constructive semantics, can carry out such analyses. Our method is illustrated by a zFIFO (0 fall-through time FIFO queue), which is a system envolving numerous simultaneous actions whose execution order may be critical.*

MOTS-CLÉS : *systèmes réactifs, synchrone,* ESTEREL, SYNCCHARTS, *sémantique constructive, vérification de modèle.*

KEYWORDS: *reactive systems, synchrony,* ESTEREL, SYNCCHARTS, *constructive semantics, model checking.*

1. Introduction

Les méthodes de conception et de programmation synchrones ont montré leur intérêt dans les applications à contrôle dominant et dans lesquelles les qualités de sûreté sont primordiales. L'approche synchrone repose sur la notion d'*instant*. Elle adopte une modélisation des systèmes en temps discret. Le système évolue par *réactions* et chaque réaction est associée à un instant. Le comportement du système, pour une séquence d'événements d'entrée donnée, est alors une séquence (ordre total) de réactions complètement déterminée. Un langage impératif synchrone comme ESTEREL [BD91] introduit en plus une notion d'ordre causal *dans* l'instant : les actions élémentaires qui composent la réaction sont *partiellement ordonnées*. Le niveau d'atomicité de la réaction autorise la diffusion de signaux, voire des dialogues protocolaires simples dans l'instant. Ceci est un point clé qui explique à la fois la puissance et la simplicité du langage.

Il est devenu classique pour un programme synchrone de vérifier les propriétés temporelles sur le temps logique (propriétés entre instants). Dans le cas d'ESTEREL l'outil XEVE [Bou98] permet une vérification symbolique du modèle (symbolic model-checking). Il est ainsi possible d'établir formellement des propriétés de sûreté. Ces propriétés sont soit du type combinatoire, par exemple une exclusion de deux actions à tout instant, soit du type séquentiel en ce qui concerne les propriétés temporelles. Dans les deux cas les réactions sont considérées comme atomiques.

L'analyse des comportements au sein d'une réaction (propriétés intra-instant) est un domaine beaucoup moins exploré, qui a, jusqu'à présent, concerné essentiellement les concepteurs de compilateurs de langages synchrones. Un problème central est celui des *cycles de causalité*. Ce problème est inhérent à l'approche synchrone : il est tout à fait possible de faire plusieurs actions simultanées, mais tout ordre d'exécution n'est pas forcément acceptable. Certains ordres sont à rejeter car ils conduisent à des incohérences (par exemple présence et absence simultanée d'un même signal), d'autres sont choquants car ils violent le principe de causalité : l'effet précède la cause au sein de la réaction. Afin de bien cerner les problèmes liés à la notion de causalité, G. Berry a défini une sémantique dite constructive [Ber96] pour le langage ESTEREL. Nous nous appuyons sur ces résultats pour notre proposition d'analyse de propriétés *dans* une réaction.

Pourquoi attacher de l'importance à l'ordre d'exécution des actions lors d'une réaction (instantanée) ? L'intérêt d'une telle étude est évident lorsque les réactions affectent des variables ou appellent des fonctions ou procédures, qui sont, rappellons-le, exécutées en temps nul. La lecture et écriture simultanées d'une mémoire est un cas typique : suivant l'ordre d'exécution les résultats sont différents. Dans un souci de cohérence et de sûreté du code produit, il faut que le concepteur puisse vérifier que l'ordre d'exécution choisi par le compilateur respecte bien la sémantique de son application [1]. Pour prouver nos propriétés de causalité, nous suivons une approche classique

1. Il s'agit de la sémantique de son application, pas celle du langage. Nous faisons l'hypothèse que le compilateur engendre bien un code conforme à la sémantique constructive.

par réfutation, en ajoutant explicitement des causalités qui entreraient potentiellement en conflit avec celles du programme.

Plan

Nous adoptons pour cette présentation le plan suivant :

– Dans une première section, nous montrons sur un cas simplifié l'intérêt et les difficultés posées par la prise en compte d'évolutions parallèles en présence de préemptions et de réactions immédiates.

– Nous expliquons ensuite comment la sémantique constructive peut donner un sens non ambigu à ces comportements réactifs complexes.

– La troisième partie illustre sur un exemple concret comment pouvoir bénéficier des possibilités offertes par la programmation réactive synchrone sans tomber dans le piège des cycles de causalité. L'exemple choisi est une z-FIFO (zero fall-through time FIFO queue), c'est à dire une file gérée avec une politique premier entré, premier sorti et qui de plus permet des dépôts et retraits simultanés même en cas de vacuité ou de saturation de la file.

2. Comportements réactifs complexes

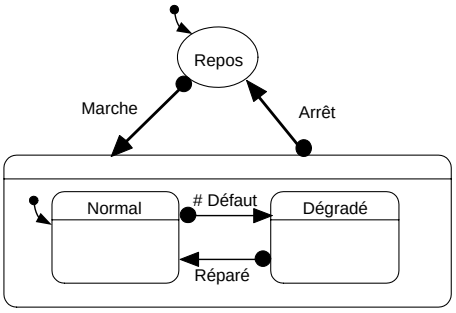

FIG. 1. *Un contrôleur simple.*

Le syncChart de la figure 1 représente le contrôle de haut niveau d'un système ayant un mode `Repos`, un mode `Normal` et un mode `Dégradé`. Le passage en mode `Dégradé` est immédiat sur occurrence du signal `Défaut`. Les SYNCCHARTS [And96] sont un modèle synchrone graphique proche des Statecharts de D. Harel [HP85], mais ils sont plus riches dans l'expression des préemptions. De plus, les SYNCCHARTS font la distinction entre les occurrences immédiates (signe # préfixant une événement déclencheur) et occurrences différées (ou strictement futures) d'événements.

Des compilateurs de SYNCCHARTS transforment le modèle en un programme ESTEREL sémantiquement équivalent. La figure 2 est le squelette du code résultant d'une telle traduction automatique. D'autres traductions automatiques, éventuellement plus

efficaces, sont possibles. Celle présentée est toutefois parfaitement correcte. Dans le processus de traduction, des signaux locaux (`goto1`, `goto2`, `goto3`) sont introduits. Rappelons que le signe % débute un commentaire qui s'étend jusqu'à la fin de la ligne.

```
1     ... % partie déclarative
2     signal goto1, goto2, goto3 in
3       emit goto1; %état initial
4       loop % faire sans fin
5          present goto1 then
6             await Marche; emit goto2
7          end present
8          ||
9          present goto2 then
10            abort
11               run Normal % mode Normal
12            when
13               case Arrêt do emit goto1
14               case immediate Défaut do emit goto3
15            end abort
16         end present
17         ||
18         present goto3 then
19            abort
20                  run Dégradé % mode dégradé
21            when
22               case Arrêt do emit goto1
23               case Réparé do emit goto2
24            end abort
25         end present
26         ||
27         pause
28      end loop
29    end signal
```

FIG. 2. *Squelette du programme Estérel.*

Le programme figure 2 comporte à la fois

– une boucle (`loop ... end loop`),

– des instructions en parallèle (||),

– des préemptions (`await`, `abort ... when`)

– et des prises en compte immédiates de signaux (`immediate Défaut`).

Ce code, engendré automatiquement, peut dérouter un programmeur non spécialiste du langage. Le comportement de ce programme n'est pas évident. A titre d'exemple, considérons que le système soit dans le mode `Repos`. Le contrôle est alors sur le `await` ligne 6. Les autres branches du parallèle sont terminées. Supposons maintenant qu'il y ait occurrence du signal `Marche` alors que le signal `Défaut` est présent.

Le contrôle passe le `await` et émet le signal `goto2` (ligne 6), sort du `present` (ligne 7), termine le parallèle et atteint la fin de boucle (ligne 28). Le contrôle revient instantanément au début de la boucle (ligne 4) et lance les 4 branches parallèles. Le `present` ligne 9 laisse passer le contrôle au `abort` (ligne 10). Celui-ci est immédiatement préempté (ligne 14) et le signal `goto3` est émis. La branche parallèle associée termine (lignes 15 et 16). La présence du signal `goto3` fait que le `present` ligne 18 passe le contrôle au mode `Dégradé` (lignes 19 et 20). Le `pause`, ligne 27, bloque la quatrième branche du parallèle. Le signal `goto1` ne pouvant pas être émis, le `present` de la ligne 5 passe le contrôle à sa clause `else` implicite et donc termine le `present` (ligne 7), terminant ainsi la première branche du parallèle.

On constate sur cet exemple d'évolution que les chemins de contrôle sont complexes et induisent un ordre (partiel) sur les actions effectuées dans l'instant. La sémantique constructive de G. Berry permet de *construire* effectivement un ordonnancement statique correct pour toute réaction, ou bien de démontrer qu'il n'en existe pas (programmes cycliques non causaux). Dans la section suivante nous nous limiterons à expliquer les bases de cette sémantique et montrer comment elle induit un ordre partiel d'actions. Au préalable soulignons l'intérêt et la puissance du modèle utilisé (figure 1). Il supporte la hiérarchie (macro-états `Normal`, `Dégradé`). La préemption permet de mettre fin instantanément à certains comprtements pour adopter d'autres comportements (par exemple, passage du mode `Normal` au mode `Dégradé`). Le fait d'être capable de prendre en compte des occurrences immédiates (nuance non supportée par les Statecharts) permet d'éliminer, par compilation, des comportements transitoires qui pourraient être préjudiciables au système. Une possibilité analogue existe dans le Grafcet interprété avec recherche de stabilité : lors d'une réaction les étapes intermédiaires instables ne sont pas visibles au niveau du comportement externe. Un dernier avantage est que la traduction des SYNCCHARTS vers des programmes ESTEREL est une traduction structurelle ; le raffinement d'un macro-état ne remet pas en cause les traductions déjà faites.

En résumé, les SYNCCHARTS permettent de tirer le meilleur parti des hypothèses synchrones. Ils expriment des comportements réactifs complexes, qui restent *déterministes* même en présence de parallélisme et de préemptions. Comme l'a montré l'exemple précédent, des comportements transitoires peuvent être complétement éliminés par la compilation, ce qui conduit à du code plus efficace et plus sûr (les transitoires sont particulièrement délicats à gérer dans les approches asynchrones). Il existe tout de même une difficulté dans l'utilisation des SYNCCHARTS ou du langage ESTEREL : il est possible, lorqu'on combine boucles, parallélisme et préemptions d'obtenir des *cycles de causalité*. Ces cycles expriment des incohérences causales qui font qu'il est impossible de trouver pour chaque état et pour chaque entrée une et une seule réaction. Le programme est alors rejeté. La connaissance, même naïve, de la sémantique constructive est nécessaire pour éviter ce piège.

3. L'approche constructive

3.1. *Généralités*

En ESTEREL les signaux sont les uniques supports de communication. Les signaux sont *diffusés instantanément* et donc perçus de la même façon par toute partie du programme qui est réceptive à ces signaux. A chaque instant, un signal doit être soit présent, soit absent. L'état de présence d'un signal est soit imposé par l'extérieur (pour les signaux d'entrée), soit déterminé par le programme (signaux de sortie et signaux locaux). Les signaux peuvent également porter des valeurs typées. Pour simplifier la présentation nous nous limitons aux programmes ESTEREL purs, c'est à dire les programmes ne contenant que des signaux non valués, appelés aussi signaux purs.

Soit $\mathcal{I}$ (resp., $\mathcal{O}$) la sorte d'entrée (resp., de sortie) d'un programme ESTEREL pur. On appelle *événement d'entrée* (resp., *événement de sortie*) le sous ensemble des signaux d'entrée (resp., de sortie) présents dans l'instant. Soit $\mathcal{L}$ l'ensemble des signaux locaux au programme.

Soit $I : \mathcal{I} \rightarrow \{0, 1\}$ le vecteur booléen associé à un événement d'entrée, tel que $\forall s \in \mathcal{I}, I(s) = 1$ si et seulement si s est présent dans l'instant. On définit de façon analogue les vecteurs booléens O, L et X (vecteur caractéristique des points d'arrêt actifs). Pour un état donné du programme, pour un événement d'entrée donné, il faut déterminer l'événement de sortie, l'état des différents signaux locaux et l'état suivant. Un programme est dit *réactif* quand il existe toujours au moins une solution. Un programme est dit *déterministe* quand il y a au plus une solution. Les programmes ESTEREL doivent être à la fois réactifs et déterministes.

La sémantique du langage est une sémantique de point fixe. Les instructions d'un programme ESTEREL pur peuvent être exprimées sous forme de circuits, et donc correspondent à des équations. A cause des boucles, mais aussi à cause du fait que les signaux locaux et les signaux de sortie participent directement à la réaction, un programme ESTEREL pur peut être considéré comme un ensemble d'équations booléennes :

$$\begin{aligned} O &= F(X, I, O, L) \\ L &= G(X, I, O, L) \\ X' &= H(X, I, O, L) \end{aligned}$$

Il s'agit donc de trouver des solutions en point fixe. Le problème est que les fonctions F, G, H n'ont pas les propriétés de monotonie qui assureraient l'existence de points fixes.

Lorsque pour un programme donné, pour tous ses états, et pour tous les événements d'entrée il existe une et une seule solution au système associé, on dit que le

programme est *logiquement correct*. S'en tenir à cette propriété n'est pas entièrement satisfaisant car des solutions logiquement correctes peuvent nécessiter des interprétations non naturelles. Par exemple, dans le programme Figure 3, si `Oui` et `Non` sont des signaux de sortie, il existe une et une seule solution logique : `Oui` présent, `Non` absent, `S` présent. Mais cette solution est choquante car le test de présence effectué à la ligne 2, a exploité l'émission du signal `S` (ligne 7). Ceci n'est pas cohérent avec la sémantique du point virgule, qui en ESTEREL, exprime la séquentialité. Depuis la version 5 du langage, un tel programme est rejeté comme étant non constructif.

```
1   signal S in
2        present S then
3            emit Oui
4        else
5            emit Non
6        end present ;
7        emit S
8   end signal
```

FIG. 3. *Programme non constructif.*

Pour montrer qu'un programme est logiquement correct, on peut utiliser un *raisonnement spéculatif*. Lors d'un test de présence, si l'état de présence n'est pas connu, on peut envisager les deux hypothèses en espérant en confirmer une et infirmer l'autre plus tard. Avec la sémantique dite *constructive* du langage ESTEREL on ne propage que des *faits*, on ne fait jamais d'hypothèses. Si en propageant uniquement des faits on arrive à trouver l'état de tous les signaux du programme, on a alors construit une solution acceptable. Un programme constructif en tout état accessible et pour tout événement d'entrée est dit *correct* pour cette sémantique constructive. Dans le cas contraire, le programme est rejeté par le compilateur qui implémente cette sémantique.

Pour les détails sur cette sémantique, nous renvoyons au texte "*The Constructive Semantics of Pure Esterel*" [Ber96]. Nous allons simplement montrer comment on peut propager des faits (positifs ou négatifs) dans le cas des tests de présence. Cette illustration est importante à comprendre, car elle sera mise à profit dans la section suivante.

Soit l'instruction

```
present S then statement1 else statement2 end present
```

1) Si `S` est (certainement) présent alors `statement1` *doit* (must) être exécuté et `statement2` *ne peut pas* (cannot) être exécuté dans l'instant.

2) Si `S` est (certainement) absent alors `statement2` *doit* (must) être exécuté et `statement1` *ne peut pas* (cannot) être exécuté dans l'instant.

3) Si l'état de présence de `S` n'est pas encore connu, *rien* ne peut être dit à propos de `statement1` et `statement2`. Le compilateur doit explorer d'autres parties du pro-

gramme dans l'espoir d'obtenir une certitude sur la présence ou l'absence de `S`. S'il n'obtient pas cette information, le programme est dit non constructif et donc rejété.

3.2. *Réfutation de causalité*

Considérons deux signaux a et b dans un programme constructif P. Considérons une relation de dépendence dans l'instant $\sqsubset$. Soit P' le programme obtenu à partir de P et instrumenté de façon à imposer $a \sqsubset b$. L'instrumentation consiste à garder les émissions de b par un test passant si l'état de présence de a est connu lors de l'émission de b. En pratique, on peut remplacer dans P "`emit b`" par "`present a then emit b else emit b end present`".

Si P' n'est pas constructif, alors que P l'était, c'est qu'il existe dans P au moins une réaction contenant a et b dans une dépendence inverse de celle imposée. Si au contraire, P' reste correct (i.e., est constructif), c'est que dans P toute transition impliquant la partie de code modifiée, a et b sont dans l'une des relations suivantes :

– a et b ne sont pas simultanément présents dans la réaction,

– a et b sont simultanément présents dans la réaction et $a \sqsubset b$,

– a et b sont simultanément présents dans la réaction et indépendents (noté $a \smile b$).

La mise en œuvre de cette méthode est illustrée dans l'exemple étudié en section 4.4.2.

4. Exemple : zFIFO

4.1. *Description informelle*

Une zFIFO (0 fall-through time FIFO queue, en anglais) est un tableau de cellules mémoire avec possibilités de décalages. Contrairement aux registres à décalage ordinaires, les informations vont directement dans la cellule vide la plus proche de la sortie, si bien que lorsque la zFIFO est vide, l'information déposée est *immédiatement* disponible en sortie (d'où le qualificatif de 0 fall-through time).

Les opérations de dépôt (`Put`) et de retrait (`Get`) peuvent être exécutées simultanément, ce qui conduit à des possibilités inexistantes dans les FIFOs ordinaires, comme par exemple le dépôt et le retrait simultanés dans une file, même si cette file est vide ou pleine.

4.2. *Spécification formelle*

Considérons des zFIFOs acceptant des objets de type T. Notons $f_{n,k}$ une zFIFO de taille n et dont k cellules sont occupées. Numérotons les cellules de façon à ce que

pour $f_{n,k}$ avec $k > 0$, les cellules 0 à $k-1$ soient occupées. $f_{n,0}$ est une zFIFO vide ; $f_{n,n}$ est une zFIFO saturée. Notons enfin $f_{n,k}(j)$ le contenu de la cellule j.

On définit deux opérations sur les zFIFO :

– append : zFIFO $\times\, T \rightarrow$ zFIFO telle que si $k < n$ alors $\mathrm{append}(\mathrm{f}_{n,k}, v)$ est $\mathrm{f'}_{n,k+1}$ avec $\mathrm{f'}_{n,k+1}(k) = v$ et $\forall j : 0..k-1, \mathrm{f'}_{n,k+1}(j) = \mathrm{f}_{n,k}(j)$.

– shift : zFIFO $\rightarrow$ zFIFO telle que si $k > 0$ alors $\mathrm{shift}(\mathrm{f}_{n,k})$ est $\mathrm{f'}_{n,k-1}$ avec $\forall j : 0..k-2, \mathrm{f'}_{n,k-1}(j) = \mathrm{f}_{n,k}(j+1)$.

Considérons une zFIFO comme un système réactif ayant deux entrées `Put:T` et `Get`, qui déclenchent les opérations de même nom et deux sorties `Taken` et `Got:T` associées respectivement au succès des opérations `Put` et `Get`.

Le comportement de la zFIFO peut être spécifié par les règles de ré-écriture ci-dessous. L'ensemble des signaux d'entrée présents est écrit sous la flèche de transition, l'ensemble des signaux émis est écrit au-dessus de la flèche.

$$0 < k \leq n \Longrightarrow f_{n,k} \xrightarrow[\{Put(v), Get\}]{\{Got(f_{n,k}(0)), Taken\}} \mathrm{append}(\mathrm{shift}(f_{n,k}), v) \quad (1)$$

$$f_{n,0} \xrightarrow[\{Put(v), Get\}]{\{Got(v), Taken\}} f_{n,0} \quad (2)$$

$$0 \leq k < n \Longrightarrow f_{n,k} \xrightarrow[\{Put(v)\}]{\{Taken\}} \mathrm{append}(f_{n,k}, v) \quad (3)$$

$$f_{n,n} \xrightarrow[\{Put(v)\}]{\emptyset} f_{n,n} \quad (4)$$

$$0 < k \leq n \Longrightarrow f_{n,k} \xrightarrow[\{Get\}]{\{Got(f_{n,k}(0))\}} \mathrm{shift}(f_{n,k}) \quad (5)$$

$$f_{n,0} \xrightarrow[\{Get\}]{\emptyset} f_{n,0} \quad (6)$$

4.3. *Réalisation*

Nous avons choisi de concevoir le système réactif zFIFO à l'aide de syncCharts et de code ESTEREL. L'application consiste en un ensemble de cellules mémoires interconnectées. Nous avons défini trois classes principales. Il s'agirait plus exactement de ≪capsule≫ dans la terminologie UML :

– La mémoire double accès (Dual-Access Elementary Memory, ou DAEM). Ses responsabiltés sont la mémorisation et la gestion d'accès. Les ports d'entrée sont `Load` et `Store`, les ports de sortie `Written`, `Read` et `Occupied`. Notre approche étant synchrone, les méthodes peuvent être sollicitées simultanément. Un syncChart définit le modèle dynamique des objets de cette classe (Figure 4).
– La *Cellule* qui est composition d'un DAEM et d'un contrôleur dédié. Ce dernier assure les communications entre cellules voisines. La programmation a été faite en ESTEREL.
– La zFIFO est une agrégation de Cellules. Le nombre n de Cellules est une constante donnée.

Dans le cadre de cette présentation, nous ne détaillons pas les programmes eux-mêmes.

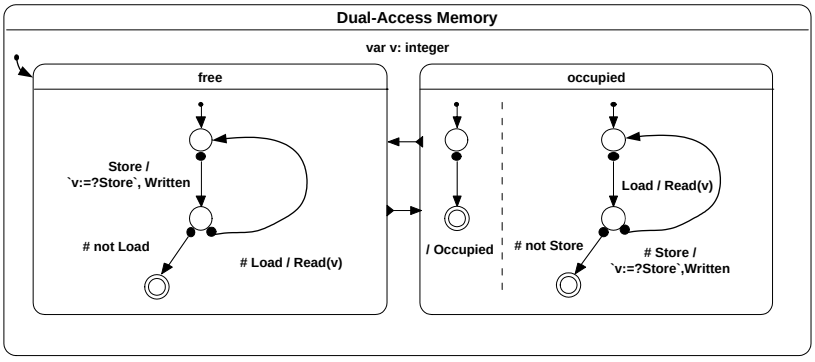

FIG. 4. *Comportement d'une mémoire élémentaire à double accès.*

4.4. *Vérification formelle*

Pour établir la correction de la solution proposée, nous avons procédé en deux temps :

1) Validation du comportement temporel essentiellement entre instants,

2) Validation de l'ordre des actions *dans* l'instant.

Pour la première phase, nous avons utilisé la technique, maintenant bien établie en programmation synchrone, qui consiste à composer le programme à vérifier avec des observateurs [Hal93]. Les observateurs caractérisent des propriétés de sûreté qui sont vérifiées par un "symbolic model-checker". Dans notre cas, ces propriétés vont être utilisées comme des préalables à l'établissement de la propriété ci-dessous.

La seconde phase consiste à montrer que *dans* toute réaction, l'ordre d'exécution des opérations concurrentes de lecture et écriture dans les diverses cellules, respecte la

discipline FIFO (premier entré, premier sorti). C'est lors cette phase que nous utilisons la sémantique constructive.

Lors d'une réaction, une cellule peut est lue (signal `R`) par un `Load` ; écrite (signal `W`) par un `Store` ; lue puis écrite (signal `RbW`, Read before Written) ou bien écrite puis lue (signal `WbR`, Written before Read). Les signaux `R`, `W`, `RbW`, `WbR` ont été rajoutés par permettre les observations.

4.4.1. *Propriétés établies par observation temporelle*

Avec l'aide de l'outil XEVE nous avons établi plusieurs propriétés. Les deux premières concernent le fonctionnement des Cellules.

P1 : Pour toute Cellule et à tout instant, `R`, `W`, `RbW` et `WbR` sont exclusifs.

P2 : Pour chaque Cellule, les traces d'exécution sur {`R`,`W`,`RbW`,`WbR`} sont
- préfixes de (`WbR`*;`W`;`RbW`*;`R`)* pour la Cellule d'indice 0 ;
- préfixes de (`W`;`RbW`*;`R`)* pour les Cellules d'indice $j > 0$.

Ces propriétés garantissent qu'il n'y a jamais perte d'information par écrasement. De plus, **P2** assure qu'une valeur lue a nécessairement été écrite avant. Remarquons que "avant" signifie soit dans un instant antérieur, soit précédemment au cœur de la réaction. Ce dernier type de propriété est établi au prix d'une instrumentation du code (introduction des signaux `RbW`, `WbR`).

Les cinq propriétés suivantes portent sur le comportement d'une zFIFO $f_{n,k}$

P3 : Pour toute Cellule d'indice j, $0 < j < n$, $\texttt{Occupied}[j] \implies \texttt{Occupied}[j-1]$.

P4 : Pour toute Cellule d'indice j, $0 \leq j < l < k$, $\texttt{Occupied}[l]$ and $\texttt{Get} \implies \mathrm{RbW}[j]$.

P5 : Un objet ne peut être retiré que si et seulement si `Get` est présent et, soit la file n'est pas vide, soit la file est vide mais `Put` est présent.

P6 : Un objet ne peut être déposé que si et seulement si `Put` est présent et, soit la file n'est pas pleine, soit la file est pleine mais `Get` est présent.

P7 : not Occupied[0] and Put(v) and Get $\implies$ WbR[0]

La propriété **P3** est une propriété structurelle qui montre que les informations sont mémorisées dans les Cellules d'indice 0 à $k-1$, avec k le nombre de Cellules occupées. La propriété **P4** exprime le décalage de l'information lors d'un `Get`. Les propriétés **P5** et **P6** sont typiquement le comportement attendu pour les opérations de retrait et de dépôt. Elles impliquent également que les spécifications 4 et 6 sont satisfaites. La propriété **P7** établit que la spécification 2 est satisfaite.

Les autres spécifications (1,3,5) qui font référence aux fonctions `append` et `shift` ne peuvent pas être directement vérifiées par XEVE qui sait traiter les booléens mais pas des types arbitraires T. Leur vérification est l'objet de la deuxième phase.

4.4.2. *Propriété établie par analyse causale*

Montrer que l'ajout d'information se fait toujours en "queue" de la zFIFO lors d'une opération de dépôt (`Put`) est assez facile. Il s'agit en fait d'une propriété de sûreté qui peut être établie par XEVE en introduisant quelques signaux auxiliaires pour l'observation.

Seule l'opération de `shift` pose vraiment problème. Notons r_j (respectivement, w_j) l'action de lecture (respectivement, d'écriture) dans la Cellule j. Soit $\prec$ la relation d'ordre dans un instant. Les propriétés précédentes et la propriété **P4** en particulier, nous permettent d'assurer que "pour toute Cellule j occupée, autre que la dernière, $r_j \prec w_j$ lors de l'exécution d'un `Get`".

Par contre, comme les Cellules évoluent *en parallèle* (elles sont composées par l'opérateur $||$ en ESTEREL), il n'est pas évident que la sémantique de l'opération de décalage (`shift`) soit respectée. Il faut arriver à montrer que lors d'une opération de `Get`, pour toute Cellule d'indice j occupée, on respecte la propriété suivante :

$$\forall j, 1 \leq j < k, r_j \prec w_{j-1}$$

C'est à dire que l'information qui est écrite dans la Cellule $j-1$ a été au préalable lue dans la Cellule j.

Pour établir cette propriété remarquons tout d'abord que l'action r_j (resp., w_j) ne peut être exécutée que par une instruction appartenant au corps du module associé à la Cellule j. Notre approche par ≪capsule≫ a largement contribué à cette "localisation" des actions. Pour être plus explicite, regardons un extrait du code décrivant le comportement d'une Cellule occupée (Figure 5).

```
1 ...% in Cell[j]
2 trap occupied in
3      loop
4          await Load;
5          % notre point de test
6          emit Read(v_j); % r_j
7          present Store then
8              v_j :=?Store % w_j
9          else
10              exit occupied
11          end present
12      end loop
13 end trap
14 ...
```

FIG. 5. *Extrait du code de la DAEM.*

Au commentaire de la ligne 5 nous substituons l'instruction ESTEREL

```
present Check then nothing else nothing end present ;
```

`Check` est un nom de signal qui sera remplacé par un véritable signal lors de l'instanciation du module. Par la séquentialité des instructions (5) et (6), on impose la relation de causalité supplémentaire "$Check \prec r_j$".

Nous effectuons deux compilations en prenant successivement pour $Check$ les signaux r_{j+1} et r_{j-1}. Le premier programme ainsi instrumenté se révèle être constructif, alors que le second ne l'est pas.

Du premier test on déduit que dans le programme non instrumenté, pour toute transition émettant r_j, on a
- soit r_{j+1} et r_j non simultanés,
- soit r_{j+1} et r_j simultanés et $r_{j+1} \prec r_j$,
- soit r_{j+1} et r_j simultanés et $r_{j+1} \smile r_j$.

Du second test on déduit que le programme non instrumenté contient une relation qui entre en conflit avec $r_{j-1} \prec r_j$. De la conjonction des deux conclusions et en tenant compte des indices on déduit que nécessairement le programme original imposait $r_{j+1} \prec r_j$, lorsque ces émissions étaient simultanées. C'est en particulier le cas lors d'un `shift`.

La relation $r_{j+1} \prec r_j$ tient pour $0 \leq j < k-1$, où k est le nombre de Cellules occupées. Comme par ailleurs, nous avons déjà montré que $r_j \prec w_j$ pour une cellule occupée, en présence de `Get`, on en déduit que $r_{j+1} \prec w_j$ pour $0 \leq j < k-1$, soit $r_l \prec w_{l-1}$ pour $1 \leq l < k$, ce qui est la relation d'ordre qu'il fallait garantir pour assurer un décalage correct.

On constate qu'en fait, le compilateur a imposé un ordre des actions plus fort $r_{j+1} \prec r_j$ alors qu'il suffisait d'avoir $r_{j+1} \prec w_j$. Cette solution est donc acceptable. Il est clair que l'utilisateur peut difficilement déterminer lui-même sur un programme un peu complexe, l'ordre des actions dans l'instant. Le compilateur est fait pour ceci. Notre technique a donc consisté à utiliser le compilateur comme un vérificateur de contrainte.

5. Conclusion

L'étude des zFIFOs et de leur programmation en SYNCCHARTS et ESTEREL qui illustre cette présentation, nous a permis de mettre en avant plusieurs points forts de l'approche synchrone, mais aussi d'analyser certaines des difficultés qu'elle pose.

En ce qui concerne la *modélisation* et l'expression des comportements réactifs complexes, l'association SYNCCHARTS, ESTEREL est particulièrement efficace. Les SYNCCHARTS encouragent les descriptions en terme d'états et de transitions entre états. Mais il ne s'agit pas de simples machines de Mealy : les SYNCCHARTS acceptent des représentations hiérarchiques, supportent les évolutions parallèles et offrent plu-

sieurs types de préemption. Il est également apparu que les SYNCCHARTS permettaient d'exprimer des comportements transitoires conceptuels. Contrairement aux solutions logicielles asynchrones, ces transitoires sont éliminés par la compilation. Les solutions synchrones passent directement d'un état stable à l'état stable suivant, par définition même des réactions synchrones. Tous les comportements ne s'expriment pas forcément facilement avec un modèle graphique, une expression textuelle est alors préférable. Les plates-formes de développement comme Esterel Studio [2] permettent de combiner les deux modèles. Quoi qu'il en soit, il faut se souvenir qu'un syncChart est traduit en un programme ESTEREL sémantiquement équivalent, si bien que la sémantique du langage ESTEREL reste la référence incontournable pour les problèmes de génération et de validation.

Les langages synchrones ont été créés pour permettre une programmation sûre des systèmes réactifs. Nous avons indiqué les vérifications formelles qui ont été effectuées sur la zFIFO. La vérification de propriété de sûreté par "symbolic model-checking" est maintenant courante. Nous avons mentionné les propriétés établies, sans les détailler. Ces propriétés sont souvent des propriétés temporelles qui s'appliquent sur les histoires d'entrées-sorties. Dans le cas de la zFIFO, les résultats obtenus étaient déjà très intéressants, mais il manquait une propriété essentielle : montrer que la politique premier entré, premier sorti était garantie. Pour celà, nous avons été amenés à introduire un nouveau type de propriétés : l'*ordre partiel d'exécution des actions dans un instant*. Pour que la discipline FIFO soit respectée, il faut que les actions de lecture et écriture simultanées, respectent une relation d'ordre partiel. Nous avons lié cette propriété à la notion de causalité et nous avons utilisé le compilateur lui-même pour tester l'ordre d'exécution réel. Ce résultat est à notre connaissance, le premier du genre.

Le travail présenté n'est qu'une étape et nous comptons bien le prolonger.

- Les *propriétés dans l'instant* semblent ouvrir une voie de recherche intéressante. De plus, pour les applications critiques qui adoptent une approche synchrone, leur vérification pourrait devenir incontournable.
- Les *aspects méthodologiques*. La conception de la zFIFO et les vérifications de ses propriétés, ont largement bénéficié de l'approche modulaire adoptée. Nous inspirant des approches objet, nous avons conçu l'application comme un ensemble de capsules coopérantes. Il ne s'agit toutefois pas d'une mise en œuvre de méthodologie comme ROOM [SGW94]. Nous avons gardé les spécificités de l'approche synchrone, en particulier les possibilités de diffusion et de dialogues instantanés. C'est un de nos objectifs, au travers d'exemples multiples et variés de définir une méthodologie adaptée aux approches synchrones.
- Pour les *méthodes de preuves*, nous avons pu, dans cet exemple, nous reposer sur le compilateur lui-même. Ceci ne nous a pas beaucoup surpris car étant basés sur des modèles mathématiques simples, les méthodes de compilation et d'analyse de programmes synchrones sont souvent très proches. Toutefois, des *proprié-*

2. http ://www.esterel-technologies.com

tés dans l'instant, en particulier celles impliquant des opérations sur données, restent encore à découvrir.

6. Bibliographie

[And96] C. André. Representation and analysis of reactive behaviors : A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

[BD91] F. Boussinot and R. De Simone. The ESTEREL language. *Proceeding of the IEEE*, 79(9) :1293–1304, September 1991.

[Ber96] G. Berry. *The Constructive Semantics of pure Esterel*. not yet published, available on the web, `www.inria.fr/equipes/meije/esterel`, Sophia Antipolis (F), 1996.

[Bou98] Amar Bouali. XEVE : An esterel verification environment. volume 1427, Vancouver (BC, Canada), 1998. Int'l Conf. on Computer-Aided Verification (CAV'98), LNCS. also available as a technical report INRIA RT-214, 1997.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Amsterdam, 1993.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems in logic and models of concurrent systems. *NATO ASI Series, K.R Apt Ed., Springer-Verlag*, 13 :477–498, 1985.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. J. Wiley Publ., 1994.