Comparaison des styles de programmation de langages synchrones

(version 1.0 du 25 juin 2005)

Charles André

projet AOSTE (CNRS - UNSA - INRIA)
Laboratoire I3S
Les Algorithmes - Bât Euclide B
2000 route des Lucioles – B.P. 121
F-06903 Sophia Antipolis Cedex
andre@unice.fr

RÉSUMÉ. *Les langages synchrones sont des langages spécialisés conçus pour la programmation sûre de systèmes réactifs et temps réel. Ils sont maintenant utilisés dans d'autres domaines comme la conception de haut niveau de circuits complexes et la programmation de systèmes embarqués. Les langages et formalismes synchrones s'appuient sur des modèles mathématiques. Ils se prêtent ainsi à des vérifications formelles.*
Ce rapport est une version largement étendue de l'article «Langages et formalismes synchrones – Fondements» présenté à MSR'05. Il introduit les fondements de l'approche synchrone. Un exemple simple permet de préciser syntaxe, sémantique et modèle de calculs associés à différents langages synchrones. Dans ce rapport, les exemples sont traités complètement pour les divers langages. Les codes sources et les simulations sont également fournis, permettant au lecteur de mieux comprendre les traits communs et ceux spécifiques à chaque langage.

ABSTRACT. *Synchronous Languages are special purpose languages dedicated to safe programming of real-time and reactive systems. There are now also used in high-level modeling and analysis of complex digital circuits and embedded systems. Synchronous languages and formalisms rely on sound mathematical models. Thus, they are amenable to formal verifications and analysis.*
This report is a widely extended version of the paper presented at MSR'05 and entitled "Langages et formalismes synchrones – Fondements". It introduces the foundations of the synchronous approach. The syntax, semantics, and model of computation associated with various languages are illustrated by a simple example. In this report, source codes and simulation results are provided for the various synchronous languages, so that the reader may have a better understanding of commonalities and differences of each language.

I3S - RR–2005–13., pages 1 à 30

MOTS-CLÉS : *Approche synchrone, systèmes réactifs, temps logique, programmation.*
KEYWORDS: *Synchronous approach, Reactive systems, logical time, programming.*

1. Introduction

Les *systèmes réactifs* sont des systèmes qui sont en interaction permanente avec leur environnement. Aux sollications de leur environnement, les systèmes réactifs répondent par des *réactions*. En retour, ces réactions peuvent modifier l'environnement. Les systèmes réactifs sont utilisés depuis longtemps en commande et contrôle. Les échanges d'information depuis l'environnement contrôlé vers le système de contrôle se font par l'intermédiaire de capteurs. Le cheminement inverse passe par les actionneurs. L'apparition des programmes dits réactifs en informatique est plus tardive. Elle répondait initialement à un besoin accru d'interactivité machine/opérateur. Avec l'émergence des réseaux et des *systèmes embarqués*, la maîtrise des interactions machine/machine est devenue essentielle. Dans les systèmes embarqués, les calculateurs (processeurs d'usage général ou plus souvent processeurs spécialisés) ne sont que des composants du système global. Leur présence est généralement discrète, voire ignorée de l'utilisateur. A titre d'exemple une automobile moderne contient plus d'une cinquantaine de processeurs.

Les langages synchrones

Les *langages synchrones* (Benveniste *et al.*, 1991, Halbwachs, 1993) ont été créés spécialement pour répondre aux besoins de la programmation d'applications réactives. Dans les années 1980, des études indépendantes ont conduit à l'apparition de langages spécialisés comme ESTEREL (Boussinot *et al.*, 1991), LUSTRE (Halbwachs *et al.*, 1991) et SIGNAL (Le Guernic *et al.*, 1991), ainsi qu'à celle d'un formalisme graphique appelé STATECHARTS (Harel, 1987). L'importance des interactions entre le programme et son environnement explique que ces langages aient été initialement conçus par des équipes mixtes comprenant des automaticiens et des informaticiens. Quant aux STATECHARTS, ils avaient été introduits par D. Harel pour décrire le comportement de systèmes rencontrés dans des applications avioniques. Ce modèle était destiné à des ingénieurs et des pilotes d'avions.

L'objectif initial des langages synchrones était de permettre une *programmation sûre* de systèmes réactifs souvent *critiques*[1]. Pour celà, ces langages ont adopté une approche de haut niveau qui s'appuie sur des *modèles mathématiques* simples, suffisamment expressifs et sur lesquels on peut conduire efficacement des analyses et vérifications de propriétés. Depuis leur création ces langages sont passés du statut de langages académiques à celui de *langages pour applications industrielles*. Ceci a été rendu possible par les progrès spectaculaires obtenus dans leur compilation. Cette évolution ne s'est pas faite aux détriments des fondements mathématiques des langages. Bien au contraire, elle a bénéficié de ces fondements et a permis de les enrichir. Le *champ d'application* des langages synchrones c'est également *élargi*. Le langage ESTEREL est maintenant largement utilisé en conception de circuits complexes (SoC

1. Systèmes critiques : systèmes dans lesquels les défaillances ont des conséquences graves.

– Systems on Chip). SIGNAL permet la spécification de systèmes ouverts. Un article (Benveniste *et al.*, 2003) particulièrement documenté fait l'inventaire des évolutions récentes et propose une riche bibliographie.

L'importance des modèles

Pour développer les applications complexes, la tendance est de mettre en avant les modèles et leurs transformations plutôt que les programmes (Ingéniérie des modèles – Model Driven Engineering). Le paradigme synchrone paraît particulièrement adapté à la modélisation de haut-niveau et la conception des applications embarquées modernes. La création de *modèles de calculs*[2] (MoC – Models of Computation) s'appuyant sur le paradigme synchrone et leur usage en conception et développement d'applications complexes est un sujet de recherche d'actualité.

Dans son livre (Edwards, 2000) intitulé "Languages for Digital Embedded Systems", Stephen Edwards défend l'idée que pour concevoir des applications aussi variées que celles que l'on rencontre dans les systèmes embarqués, il faut des langages (et des modèles) différents. Les langages qu'il présente dans son ouvrage sont classés en langages pour la description du matériel (Verilog, VHDL,...), langages "standard" (C, C++, Java,...), langages flots de données et langages qu'il qualifie d'hybrides, parmi lesquels on trouve ESTEREL. Nous partageons tout à fait cette analyse. Les approches multi-modèles apparaissent à présent comme incontournables. Le fait que dans cette présentation nous ne traitions que des langages synchrones ne doit en aucun cas être interprété comme la promotion d'une approche unique. Même en nous limitant aux modèles synchrones, nous retrouverons cette diversité : les différents modèles et langages synchrones sont complémentaires complémentaires à bien des égards et sont plus particulièrement adaptés à des classes d'applications.

S. Edwards insiste aussi sur le fait pour pouvoir tirer le meilleur parti d'un langage, il est utile d'avoir une bonne compréhension de la façon dont ce langage est compilé. Ceci est vrai pour les langages synchrones. Nous allons même plus loin en préconisant une bonne connaissance des modèles mathématiques sous-jacents.

Cet article traite essentiellement de l'*usage classique* des langages synchrones pour des applications réactives dans lesquelles la notion de temps global est une hypothèse raisonnable.

Contenu

Dans une première partie, les fondements de l'approche synchrone sont introduits. Les évolutions sont découpées en *réactions* successives disjointes. Une réaction transforme des informations entrantes (signaux d'entrée) en informations sortantes (si-

2. On trouve parfois l'espression "modèles d'exécution comportementale" qui correspond mieux à la réalité.

gnaux de sortie). Le comportement entrée/sortie est *déterministe*. Ce qui distingue l'approche synchrone des autres modèles d'exécution par pas c'est la façon de déterminer ces réactions. Une réaction est un ensemble partiellement ordonné d'actions (ordre causal). Les *signaux*, unique support de communication en synchrone, jouent un rôle central dans le calcul des réactions.

Nous présentons ensuite 3 langages synchrones. Pour chacun nous dégageons certains traits syntaxiques, nous donnons des éléments sur leur sémantique et nous illustrons leur usage sur un exemple de système réactif simple. Le premier langage étudié est LUSTRE, un langage synchrone déclaratif particulièrement simple. Le second est SIGNAL. Il est lui aussi déclaratif. Il est plus général que le précédent et il exploite pleinement la notion d'absence d'un signal à un instant c'est à dire lors d'une réaction. Le dernier langage présenté est ESTEREL qui adopte un style impératif.

Les extensions de l'approche synchrone sont évoquées dans une dernière partie.

2. Le modèle synchrone

2.1. *Un exemple de système réactif*

Pour illustrer notre propos nous retenons un système réactif très simple (figure 1). Ce système peut être prétexte à deux études différentes. Soit il s'agit d'un système existant, l'objectif est alors de *simuler* ou mieux d'*analyser les comportements* possibles du système. Soit ce système est à concevoir. Dans ce cas la structure montrée dans la figure peut correspondre à une décomposition fonctionnelle imposée par des contraintes de déploiement ou simplement être une décomposition conceptuelle. Les langages synchrones peuvent servir ces deux objectifs. Dans ce qui suit, nous adoptons le point de vue conception et nous utilisons différents langages synchrones pour spécifier tout ou partie du comportement de ce système.

Spécification informelle du système

Il s'agit d'une système simplifiée de comptage du temps exprimé en minutes (M) et secondes (S). L'entrée (Top) est activée par un générateur extérieur (qui dans l'application véritable devrait être périodique de fréquence 1 Hz). Deux autres entrées de commande sont disponibles : celle de remise à zéro (RAZ) et celle d'avance rapide du compteur des minutes (Avance).

Le système est intialement dans un *mode* dit *oisif*. Dans ce mode il n'affiche rien et il est insensible aux entrées Top et Avance. La première demande de remise à zéro le fait passer dans le *mode normal*, qu'il ne quitte plus. En mode normal, les valeurs des compteurs sont affichées à chaque mise à jour. Chaque occurrence de Top incrémente le compteur des secondes modulo 60 et provoque éventuellement une incrémentation du compteur des minutes, lui aussi modulo 60. Une occurrence d'Avance incrémente uniquement le compteur des minutes. Les occurrences simultanées de la retenue et

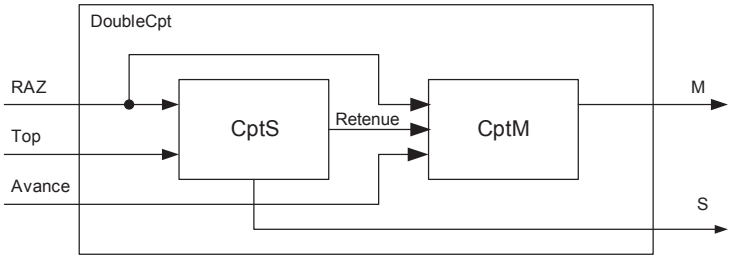

Figure 1. *Exemple : double compteur.*

d'Avance causent une incrémentation de 2 du compteur des minutes. Toute nouvelle occurrence de RAZ cause immédiatement une remise à zéro des deux compteurs.

2.2. *Caractéristiques des langages pour systèmes réactifs*

L'exemple précédent contient des traits communs à de nombreux systèmes réactifs. Les spécificités de ces systèmes font que les modèles et langages, qu'ils soient synchrones ou non, doivent répondre à des exigences particulières.

– *Interface* : la frontière entre le système et son environnement est bien définie, de même que le sens des échanges (communications entrantes ou sortantes). Les langages doivent donc permettre la *déclaration d'interfaces*. Pour les langages synchrones le *signal* est choisi comme support d'information. Les interfaces déclarent alors des signaux typés et directionnels (entrée ou sortie).

– *Décomposition en sous-systèmes* : cette décomposition peut être imposée par des contraintes au niveau de la réalisation. Elle peut également être conceptuelle et elle permet alors d'appréhender la complexité en divisant le problème. Les langages doivent offrir des facilités d'expression de décomposition et plus généralement de *structuration hiérarchique* (modules).

– *Évolutions concurrentes* : les sous-systèmes ont a priori des évolutions concurrentes. Les langages doivent exprimer le *parallélisme*.

– *Communications* : pour coopérer les sous-systèmes communiquent entre eux. Dans les langages synchrones la communication se fait par *signaux diffusés* aux différents modules.

– *Préemption* : la préemption est une forme d'interaction de contrôle. Une activité peut être suspendue ou tuée par une activité concurrente. Dans l'exemple donné, l'occurrence du signal RAZ arrête l'activité normale de comptage et la réinitialise. Les langages synchrones impératifs permettent de spécifier différentes formes de préemptions qui vont au-delà des traitements d'exception des langages classiques.

– *Promptitude* : les systèmes réactifs sont supposés réagir immédiatement aux stimuli reçus. Les spécifications de comportements temporels prennent souvent la

forme : “dès que ... faire ...”, comme c’est le cas ci-dessus pour la remise à 0 des compteurs sur l’occurrence de RAZ. Les langages synchrones adoptent ce type de comportement par défaut. Ceci n’exclut pas la possibilité d’autoriser des réactions différées avec spécification d’échéances maximales (deadlines).

– *Événements* : les informations échangées avec l’extérieur ou entre sous-systèmes peuvent être fugaces (événements) ou présenter une certaine persistance (données). Les événements se manifestent par leurs occurrences. Ils peuvent éventuellement être porteurs de valeurs. Les langages devraient permettre de distinguer ces deux types d’information.

2.3. *Langages Synchrones*

Dans la suite nous parlons de programmes et de modules. Les explications qui suivent s’appliqueraient aussi à des systèmes et sous-systèmes.

2.3.1. *Aspects statiques*

Les objets manipulés par les langages synchrones sont principalement les signaux et les actions. Nous donnons ici un sens très général à ces termes, leur définition ne coïncide pas nécessairement avec celles utilisées en ESTEREL et SIGNAL, en particulier pour le mot signal.

Un *signal* est typé. On distingue en fonction du rôle qu’ils jouent dans le programme : les signaux d’entrée (ensemble $\mathcal{I}$), les signaux de sortie (ensemble $\mathcal{O}$) et les signaux locaux (ensemble $\mathcal{L}$). Les signaux sont, dans les langages synchrones, le moyen unique de *communication*, aussi bien avec l’environnement qu’entre les unités de programmation du langage.

Les *actions* accèdent aux objets du programme et peuvent les modifier. C’est en particulier le cas pour les signaux qui sont accédés et modifiés par des actions spécifiques.

Un utilisateur ne voit comme résultat de l’exécution d’un programme synchrone que des transformations de signaux d’entrée en signaux de sortie. Ainsi présentés, les langages synchrones pourraient être assimilés à des langages de flots (*streams*). Il y a en plus une hypothèse implicite de découpage des exécutions en *réactions* successives disjointes (c’est à dire sans recouvrement d’exécutions). La notion de réaction est intimement liée à celle de système réactif. Une réaction est la réponse du système à un *stimulus* (réception d’un événement). Elle se manifeste par l’émission de signaux vers l’environnement. Cette idée de découper les évolutions en pas contrôlés n’est pas propre au synchrone. Elle est largement utilisée en mathématique et automatique. Les machines à états finis évoluent également de cette façon.

Les schémas d’exécution les plus communs pour un programme synchrone sont donnés dans la figure 2. Le schéma de gauche correspond aux réactions déclenchées par événement (event driven) ; le schéma de droite utilise un *signal particulier* appelé

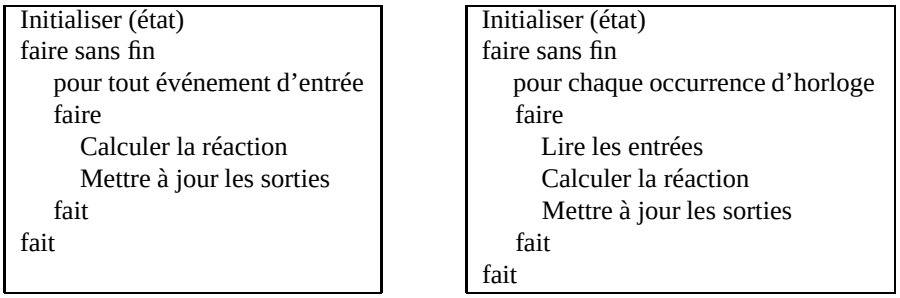

Figure 2. *Schémas d'exécution synchrones.*

horloge et échantillonne les entrées (clock driven). Ce dernier est également utilisé dans la plupart des automates programmables industriels. Dans ces schémas, "Calculer la réaction" consiste à effectuer des actions, déterminer l'état suivant du programme et les sorties en fonction de l'état courant et des entrées.

Un programme synchrone est en fait une façon de spécifier des contraintes entre les signaux. Les langages impératifs (ESTEREL et SYNCCHARTS) expriment ces contraintes par des structures de contrôle et des séquencements explicites d'actions. Les langages déclaratifs (LUSTRE et SIGNAL) ont recours à des équations qui expriment des dépendances fonctionnelles ou relationnelles.

2.3.2. *Aspects dynamiques*

Un utilisateur peut se contenter de cette vision "boîte noire" qui se limite à l'observation des entrées/sorties. D'ailleurs les relations d'entrée/sortie jouent un rôle essentiel dans la sémantique des langages synchrones (§ 2.3.5). Toutefois, cette vision externe occulte complètement l'activité qui conduit à l'émission des signaux et donc l'explication du pourquoi de ces émissions. Pour comprendre comment un système évolue ou ce que fait un programme il est préférable d'étudier son *comportement*. Pour celà nous proposons d'utiliser les concepts d'*activité* et d'*action*[3] d'UML 2, spécialisés pour l'approche synchrone. Une réaction est alors l'exécution d'une activité, c'est à dire un ensemble d'exécutions d'actions partiellement ordonnées. Cette ensemble est nécessairement fini pour une réaction synchrone. L'exécution d'une action peut réaliser des traitements sur des données et aussi émettre des signaux soit à destination de l'environnement, soit à usage interne (communications entre modules). Ces signaux peuvent à leur tour provoquer de nouvelles exécutions d'actions.

La figure 3 représente sous forme de diagramme d'activité UML la réaction du système de comptage, causée par une occurrence de Top, lorsqu'il est en mode normal

3. An *activity* is the specification of parameterized behavior as the coordinated sequencing of subordinate units whose individual elements are actions. An *action* is a named element that is the fundamental unit of executable functionality. (UML 2.0 superstructure).

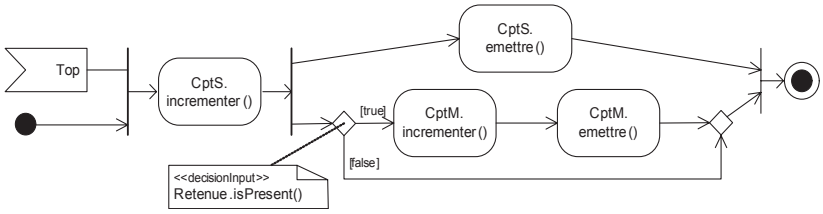

Figure 3. *Un exemple de réaction.*

et que le signal RAZ est absent. En fait, ce diagramme superpose 2 ordres partiels qui correspondent à des réactions différentes caractérisées par la présence ou non du signal Retenue.

Dire qu'une réaction est un ensemble finie d'exécutions d'actions partiellement ordonnées n'est absolument pas suffisant. Il faut pouvoir déterminer quelles sont ces actions et définir la relation d'ordre. De plus, on aimerait que la connaissance de la réaction de chaque module permette de calculer la réaction du programme. Malheureusement, la composition parallèle de deux exécutions synchrones n'est pas nécessairement synchrone. En effet, les communications entre modules peuvent créer des cycles de dépendance qui ne permettent plus d'ordonner les actions concernées. On dit alors que le système n'est plus réactif. Pour plus de détails sur ce point le lecteur est renvoyé à la référence (Benveniste *et al.*, 2003).

2.3.3. *Signaux*

Nous avons déjà mentionné que les *signaux* étaient les supports de communication. La propriété essentielle des signaux dans les approches synchrones est qu'au cours d'une réaction chaque signal peut prendre *au plus un état*. L'état peut être caractérisé par une simple présence pour les signaux représentant des événements. Un signal peut aussi être porteur d'une valeur typée. Lorsque le signal est un signal d'entrée, son état est imposé par l'environnement. En revanche, l'état des autres signaux, qu'ils soient de sortie ou locaux, est fixé par le programme.

La technique de détermination de l'état des signaux diffère entre les approches impératives à la ESTEREL et les approches déclaratives à la SIGNAL. Elles utilisent respectivement une technique de propagation d'information et un calcul d'horloge, comme nous l'expliquerons plus loin.

Cette détermination peut conduire à différentes situations :

1) tous les signaux reçoivent un état unique : la réaction est *déterministe* ;

2) tous les signaux reçoivent un état, mais plusieurs solutions sont acceptables pour certains signaux : la réaction est *non-déterministe* ;

3) on ne peut pas trouver d'états acceptables pour certains signaux : on ne sait alors pas définir la réaction.

Le diagnostic fait sur une réaction est étendu au programme lui-même. Un programme pour lequel toutes les réactions sont déterministes est dit *réactif et déterministe*.

Lorsqu'il s'agit d'engendrer un code exécutable, les seuls programmes acceptés par les compilateurs de langages synchrones sont ceux qui satisfont cette double propriété de réactivité et de déterminisme. Ces programmes sont particulièrement appréciés pour les applications réactives. Toutefois, une conséquence de ce choix est qu'un programme *syntaxiquement correct* peut être rejeté par le compilateur. Cette situation est parfois critiquée (voir par exemple (Harel *et al.*, 2000)). Les STATECHARTS, qui par bien des aspects sont proches des modèles synchrones, n'ont pas adopté ce point de vue. Un statechart syntaxiquement correct est toujours accepté mais il peut avoir un comportement non déterministe.

Les programmes synchrones non-déterministes ou non-réactifs peuvent être utiles en spécification de haut-niveau ou en conception partielle. Le compilateur SIGNAL sait en tirer parti.

Restreindre les programmes corrects à ceux qui vérifient la condition 1 est quelque peu sévère. On pourrait très bien dans certaines réactions avoir un signal local dont l'état est indéfini ou défini de façon multiple, sans qu'il ait une influence sur l'état des signaux de sortie et sur l'état suivant du programme. Des règles moins contraignantes permettraient de caractériser des classes de programmes réactifs déterministes plus grandes que celles actuellement acceptées par les compilateurs. En fait, les compilateurs, qui sont déjà bien complexes, n'exploitent que des conditions suffisantes de réactivité/déterminisme. Une discussion (Boussinot, 1998) sur des programmes rejetés par les différentes versions du compilateur ESTEREL illustre bien ces problèmes.

2.3.4. *Instant*

Chaque réaction d'un programme synchrone se produit à un *instant* particulier. La suite des instants définit un *temps logique*. La "distance" entre deux instants (durée) n'a aucune signification en temps logique, seul l'ordre importe.

La notion d'instant permet de définir clairement la *simultanéïté* comme étant la présence de signaux au même instant. Une conséquence est la possibilité de définir une notion de *priorité* objective c'est à dire qui soit significative même en présence d'évolutions concurrentes. Dans l'exemple du Comptage, le signal RAZ a priorité sur le signal Top. Ceci signifie que si les deux signaux sont simultanés, les actions déclenchées par RAZ sont exécutées prioritairement, quitte à interdire celles déclenchées par Top. Une autre conséquence est qu'il est possible de *réagir à l'absence*. Cette notion n'a de sens que sous ces hypothèses synchrones d'instant logique.

Chaque signal peut prendre au plus un état dans un instant. Déterminer les états des signaux est un problème difficile. Quelle que soit la façon de trouver ces états, la règle suivante doit être respectée : "tous les accès au statut d'un signal *dans un instant* doivent être *cohérents* (même statut de présence, même valeur)". Pour cela, il faut respecter une *relation de causalité* que l'on peut énoncer ainsi : "le statut d'un

signal doit être défini *avant* toute utilisation". La préposition *avant* exprime une précédence *dans* l'instant : au cours de la réaction, l'action qui fixe le statut d'un signal précède toute action d'accès au statut de ce signal. En SIGNAL, à partir des équations on déduit des relations de dépendance causale entre signaux (§4.2). Il faut ensuite calculer une solution point fixe. En ESTEREL, il faut tenir compte des structures de contrôle, et en particulier de la séquentialité introduite par le ' ;'. Le principe du calcul est de propager des "*facts*", c'est à dire des certitudes sur la présence ou l'absence de signaux (Berry, 2000). L'exécution de l'émission d'un signal implique sa présence. Garantir son absence est beaucoup plus délicat puisque c'est une décision globale. Les compilateurs actuels implémentent une approximation de cette règle de causalité. Les programmes non causaux sont certainement rejetés, mais certains programmes pour lesquels une sémantique raisonnable pourrait être donnée, sont également rejetés. Une discussion (Boussinot, 1998) sur des programmes rejetés par les différentes versions du compilateur ESTEREL illustre bien ces problèmes.

En résumé, un programme synchrone correct est réactif et déterministe. Les réactions sont constituées d'exécutions d'actions partiellement ordonnées. Le compilateur choisit un ordonnancement uniforme qui est un ordre d'exécution total, compatible avec les ordres partiels associés à chaque instant.

2.3.5. *Sémantique*

La sémantique donne la signification des éléments du langage. Cette signification est elle-même un élément d'un domaine bien défini appelé *domaine sémantique*. Pour les langages synchrones, le domaine sémantique est défini en termes mathématiques. Cette base formelle permet entre autres des analyses et vérifications sur les programmes. Outre le domaine sémantique, il faut définir l'*application sémantique* qui relie les expressions syntaxiques aux éléments du domaine sémantique. Un langage peut d'ailleurs avoir plusieurs sémantiques (voir par exemple les sémantiques d'ESTEREL (Berry, 1999)). Dans ce cas, les fondements mathématiques permettent de comparer précisément ces sémantiques et éventuellement prouver leur équivalence.

Il n'est pas question de donner ici les sémantiques des langages synchrones. Nous nous limitons à des généralités qui se déclinent différemment suivant les langages.

L'élément sémantique associé aux signaux (en ESTEREL et SIGNAL) ou aux flots (LUSTRE) est une séquence infinie de valeurs typées indexées par les instants logiques.

Pour un signal S notons D_S son domaine de valeurs. L'absence du signal à un instant est notée $\perp$. Soit $D_S^\perp = D_S \cup \{\perp\}$. On pose $\mathbb{I} = \prod_{S \in \mathcal{I}} D_S^\perp$ et $\mathbb{O} = \prod_{S \in \mathcal{O}} D_S^\perp$. Un programme synchrone P (qui est réactif et déterministe) dénote une *fonction* des *histoires* d'entrée dans les histoires de sortie. $[\![P]\!] : \mathbb{I}^* \to \mathbb{O}^*$. Notons bien que cette définition travaille sur des séquences de uplets, ce qui convient pour l'approche synchrone. Pour les flots de données en général il faudrait considérer des uplets de séquences, ce qui est nettement plus complexe. Un rapport sur les langages synchrones flots de données (Benveniste *et al.*, 1993) étudie plus particulièrement cette extension. Cette fonction préserve la longueur des séquences (le k^e élément de la séquence cor-

respond à la k^e réaction). En introduisant un état $\mathbb{X}$, elle peut être remplacée par deux fonctions travaillant sur les uplets et non plus les séquences d'uplets :

$\lambda : \mathbb{X} \times \mathbb{I} \rightarrow \mathbb{X}$ (fonction état suivant)

$\omega : \mathbb{X} \times \mathbb{I} \rightarrow \mathbb{O}$ (fonction de sortie).

Le domaine sémantique adapté à SIGNAL utilisé en modélisation de systèmes ouverts s'appuie sur des *relations* plutôt que sur les fonctions. Une analyse générale sur les domaines sémantiques est présentée dans (Harel *et al.,* 2000). (Benveniste *et al.,* 2003) apporte des arguments plus ciblés sur les langages synchrones.

Sémantique et comportement sont intimement liés dans un système réactif, mais ce sont deux concepts différents (Harel *et al.,* 2004). Pour un utilisateur il est plus simple de comprendre le comportement du programme lors d'une réaction, plutôt que la fonction qui donne la sémantique du programme.

3. LUSTRE

3.1. *Flots*

LUSTRE (Halbwachs *et al.,* 1991) est un langage de programmation basé sur un modèle flot de données très simple. Il est conçu pour être aisément utilisable par des ingénieurs de contrôle qui sont habitués aux systèmes d'équations (différentielles, aux différences finies, booléennes) ou des réseaux flots de données (schémas blocs, circuits logiques, ...). Ceci est encore plus vrai pour l'atelier SCADE [4] qui supporte une version graphique du langage.

Un programme LUSTRE est appelé un *nœud* (node). Il est déclaré avec des paramètres d'entrée et de sortie. Le corps du nœud définit la *relation fonctionnelle* qui existe entre les entrées et sorties. Le listing ci-dessous définit le nœud AddModulo. Ses entrées sont valuées par des entiers. Il a deux sorties : S est du type entier, R du type booléen.

```
1 node AddModulo (A,B,M:int) returns (S:int;R:bool);
2 var C:int; -- déclaration de variable locale
3 let
4   C = A + B;
5   S = C mod M;
6   R = (C >= M);
7 tel
```

Les lignes 4 à 6 sont des équations du nœud. LUSTRE manipule des *variables* et des *expressions* qui dénotent des *flots*. Un flot est une suite de valeurs d'un type donné $X = (X_1, X_2, \cdots, X_n, \cdots)$ où X_n est la valeur de X à l'instant (logique) n. De fait, un programme LUSTRE décrit un réseau d'opérateurs travaillant sur des flots. Chaque opérateur consomme des données en entrée et produit des données en sortie.

4. //www.esterel-technologies.com

Les calculs sont instantanés. Les échanges de données entre opérateurs sont également instantanés.

L'instruction de la ligne 4 se lit : le flot C *est défini par* l'expression A + B. Les opérateurs usuels (arithmétiques, booléens, de comparaison ou conditionnels) sont étendus aux flots. On a ainsi $C_k = A_k + B_k$ pour tout $k = 0, 1, \cdots$

Toute variable qui n'est pas une entrée doit être *définie une et une seule fois* en termes des autres variables par une équation. Le programme est alors un système d'équations. L'ordre des équations n'a aucune importance. Toute occurrence d'une variable peut être remplacée par son expression de définition (et réciproquement). Ainsi la variable C du nœud AddModulo peut être éliminée sans changer la sémantique du programme. Ce nœud dénote en fait deux fonctions telles que pour tout $k = 0, 1, 2, \ldots$: $S_k = (A_k + B_k) \mod M_k$ et $R_k = (A_k + B_k \geq M_k)$. Notons que les constantes représentent des flots constants : true = (true,true,...).

Le nœud LUSTRE est un élément structurant du langage. Il joue le rôle d'un opérateur défini par l'utilisateur et peut être réutilisé dans d'autres nœuds.

LUSTRE possède aussi deux *opérateurs temporels* : –> et pre qui permettent de spécifier des fonctions séquentielles. Ils sont définis respectivement par

$(X -> Y)_0 = X_0$ et $(X -> Y)_k = Y_k$ pour $k > 0$

$\mathrm{pre}(X)_0 = nil$ et $\mathrm{pre}(X)_k = X_{k-1}$ pour $k > 0$

nil représente une valeur *non définie*. L'usage combiné de pre et –> évite l'accès (interdit) à cette valeur.

Le nœud CompteurMod spécifie un compteur modulo, initialement à 0, qui prend en entrée un booléen de remise à 0 (RAZ) et deux entiers : un incrément (Incr) et le modulo (M). Ce nœud réutilise le nœud AddModulo.

```
node CompteurMod (RAZ:bool; Incr,M:int) returns (C:int;R:bool);
let
  (C,R) = (0,false) -> if RAZ then (0,false) else
      AddModulo(pre(C),Incr,M);
tel
```

On remarque que LUSTRE peut manipuler des uplets, les opérateurs s'appliquent sur chaque composante.

L'exécution d'un programme LUSTRE suit le schéma d'exécution clock-driven de la figure 2. La séquence des entrées du programme induit un temps discret qui constitue l'*horloge de base* du programme. Chaque variable ou expression prend la k e valeur de sa séquence à la k^e itération.

Si on parle en termes de signaux (§ 2.3.3), les signaux d'entrée et de sortie du programme LUSTRE sont toujours présents. Il n'y a pas à proprement parler d'événements. Pour représenter une information de nature événementielle en LUSTRE on peut prendre un signal booléen. La valeur true de ce signal à un instant est interprétée comme une occurrence de cet événement. On peut aussi définir une variable Front

booléenne qui devient true en cas de changement de valeur d'une autre variable V :
Front = **false** –> (**pre**(V) <> V);

3.2. *Horloges*

Nous avons vu que dans l'approche synchrone en général, un signal peut être absent à certains instants. C'est le cas dans le système de comptage (§ 2.1) : le compteur CptS ne doit être incrémenté qu'en présence de Top. En utilisant la convention donnée ci-dessus qui assimile un événement à un booléen, on peut programmer ainsi :

```
1 ZCptS = 0 –> pre CptS; –– CptS retardé d'un instant;init 0
2 (CptS,RS) = if RAZ then (0,false) else
3     if Top then AddModulo(ZCptS,1,60) else (ZCptS,false);
```

Cette solution consiste à "figer" (garder la valeur précédente) la variable CptS en cas d'absence de Top (ligne 3). Une autre façon de traiter ce problème en LUSTRE est d'utiliser des *sous-horloges*. En LUSTRE, tout flot booléen peut être utilisé comme horloge. L'opérateur when permet à partir d'un flot X et d'une "horloge" H de définir le flot X échantillonné par H : (X when H) qui n'est défini qu'aux instants pour lesquels H est true. Il prend alors la même valeur que X à ces instants. Il existe également l'opérateur current qui joue un rôle de "bloqueur", en donnant des valeurs à des instants pour lesquels le signal n'était pas défini mais l'horloge d'échantillonnage l'était.

Définir de nouvelles horloges induit un problème de cohérence des horloges. LUSTRE a adopté des règles suivantes :

– Les entrées sont sur l'horloge de base [5].

– Les constantes sont sur l'horloge de base.

– Pour tout opérateur op, l'expression (X op Y) a pour horloge H si X *et* Y ont pour horloge H.

– (X **when** H) est correct si X et H sont sur la même horloge. L'horloge de (X **when** H) est alors H.

– L'horloge de (**current** X) est l'horloge de l'horloge de X.

LUSTRE reste tout de même un langage synchrone mono-horloge (toutes les "horloges" sont des sous-horloges de l'horloge de base). Ces règles de cohérence sont vérifiées par le compilateur. Toutefois, pour les utilisateurs habituels de SCADE ces règles sont obscures. Il a donc été rajouté à SCADE une construction particulière appelée CondAct, pour condition d'activation. Cette construction peut s'exprimer à l'aide de when et current qui sont cachés à l'utilisateur. Le but est de n'exécuter un opérateur ou un nœud que lorsqu'une condition d'activation particulière est satisfaite. Il faut préciser le nœud à contrôler et une valeur par défaut pour les sorties. Pour un nœud N ayant pour entrées E1, E2, ..., pour sortie Z, une condition d'activation CA

5. Cette contrainte peut être relaxée *explicitement* pour certaines entrées.

et une valeur de sortie par défaut Zdefaut, le comportement est spécifié par

```
Z = if CA then -> current( N((E1,E2 ,...) when CA ) )
    else (Zdefaut -> pre Z);
```

Lucid Synchrone (Caspi *et al.*, 1998) est un langage fonctionnel qui reprend les aspects logiques (du premier-ordre) de LUSTRE et les étend à un ordre supérieur. Ce langage permet en particulier un calcul d'horloge par inférence de type à la ML, ce qui permet d'aller bien au-delà du when et du current.

En résumé, le compilateur LUSTRE vérifie qu'il n'y a pas de dépendences cycliques syntaxiques entre les variables. Par un calcul d'horloge simple il s'assure que les sous-horloges sont correctes. Il lui reste alors à donner un ordre d'évaluation causal, ce qui revient à faire un tri topologique sur les équations. Il est bien entendu que les cycles incriminés sont des *cycles dans l'instant*. Il est tout à fait possible de reboucler une variable sur elle-même mais avec un retard dans la boucle (présence d'un pre), c'est le principe même de la mémorisation.

3.3. *Programmation complète du Comptage*

RAZ, Top et Avance devraient être des événements, ils sont dans notre programme LUSTRE représentés par des booléens.

Pour gérer les modes, on écrit un nœud LUSTRE Filtre. dont le rôle est de "capturer" les occurrences de Top et Avance qui pourraient arriver en mode oisif.

```
1 -- Mode = oisif codé par false
2 -- Mode = normal codé par true
3 node Filtre (RAZ, Top, Avance: bool) returns (LTop, LAvance: bool);
4 var Mode: bool;
5 let
6   Mode = false -> if RAZ then true else pre(Mode);
7   LTop = Mode and Top;
8   LAvance = Mode and Avance;
9 tel
```

Comme nous avons choisi la convention de représenter la présence par le booléen true et que d'autre part nous codons l'état oisif par le booléen false, il suffit d'utiliser une conjonction logique (lignes 7 et 8) pour bloquer les occurrences de Top et Avance en mode oisif. Le mode lui-même est une simple mémoire (ligne 6) initialement à false (mode oisif) et qui passe à true sur la première occurrence de RAZ (c'est à dire premier passage à true de RAZ) et y reste.

Le programme de Comptage qui utilise ce nœud comme l'indique la figure 4.

Le programme complet proposé est :

```
1 const Mod = 6 -- au lieu de 60, pour faciliter la simulation
2 node COMPTAGE (RAZ, Top, Avance: bool) returns (S,M: int);
3 var
```

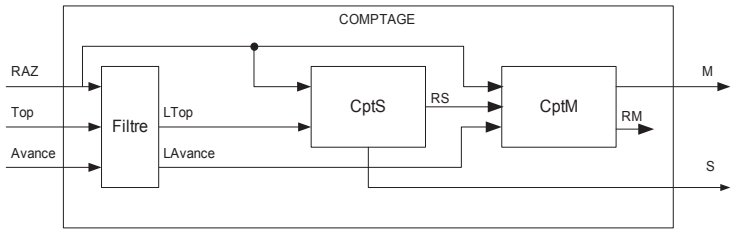

Figure 4. *Décomposition de l'application de Comptage.*

```
4    LTop,LAvance:bool; -- signaux filtrés
5    RS,RM:bool; -- retenues
6    pS,pM:int; -- valeurs précédentes
7    Incr:Incr; -- variable intermédiaire
8  let
9    pS = 0 -> pre S;
10   pM = 0 -> pre M;
11   (LTop,LAvance) = Filtre (RAZ,Top,Avance);
12   (S,RS) = if RAZ then (0,false) else
13        if LTop then AddModulo(pS,1,Mod) else (pS,false);
14   Incr = if RS and LAvance then 1 else 0;
15   (M,RM) = if RAZ then (0,false) else
16        if RS or LAvance then
17          AddModulo(pM,1+Incr,Mod)
18        else
19          (pM,false);
20 tel
```

La ligne 1 est une déclaration de constante. Mod devrait être 60. Il est pris égal à 6 pour faciliter la simulation. Le gestion de l'incrémentation du compteur des minutes est un peu compliquée par le fait qu'elle peut être causée par Avance, RS (la retenue du compteur des secondes), ou par les deux. Dans les deux premiers cas la valeur de l'incrément doit être 1 et 2 dans le troisième cas. Une solution possible est d'utiliser la variable locale Incr dont la valeur est évaluée à chaque instant (ligne 14). Une solution moins artificielle mais plus bavarde aurait consisté en des **if** ... **then** ... **else** ... imbriqués pour traduire les divers cas. Le prix à payer aurait été une formule arborescente.

LUSTRE dispose d'une sortie de simulation sous forme de chronogramme (SIM2CHRO). La figure 5 contient la trace d'une simulation.

En résumé, les instructions de LUSTRE sont très simples. C'est un avantage pour l'apprentissage du langage. Toutefois, cette simplicité entraîne des inconvénients :

– L'absence des signaux doit être gérée par l'utilisateur lui-même. Ceci peut amener à introduire des signaux supplémentaires.

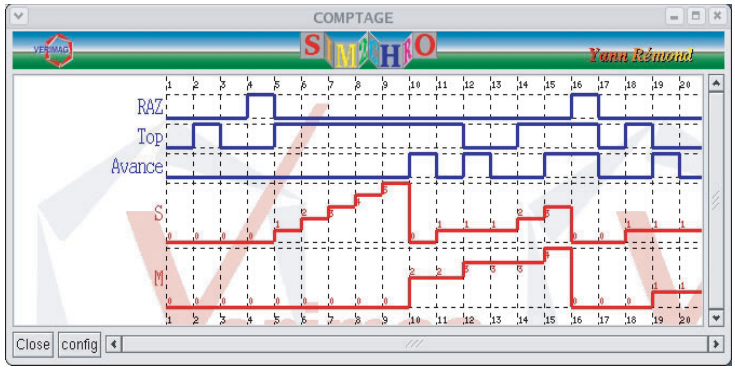

Figure 5. *Chronogrammes du programme* LUSTRE *"Comptage".*

– Le caractère équationnel de LUSTRE ne facile pas l'expression des modes, qui sont plus aisément pris en compte par des formalismes basés sur les états.

4. SIGNAL

Comme nous l'avons vu précédemment, en LUSTRE, les horloges sont simples : elles sont toutes sous-horloges de l'horloge de base. Il en est autrement en SIGNAL (Le Guernic *et al.*, 1991) qui exploite pleinement la notion d'absence de signaux. Par exemple, pour l'application de comptage, les signaux d'entrée Top et RAZ sont indépendants ; il est réducteur de vouloir les ramener à une même horloge. SIGNAL est particulièrement bien outillé pour traiter ces aspects multi-horloges.

4.1. *Concepts du langage* SIGNAL

Un programme SIGNAL manipule des séquences infinies de valeurs typées. Ces valeurs sont implicitement indexées sur un temps logique discret. Pour cette présentation, il nous suffit de considérer que le temps logique T est un ensemble d'instants totalement ordonnés. Le cas plus général considère un ensemble partiellement ordonné (Le Guernic *et al.*, 2003).

Un *signal* S est défini par une suite de valeurs typées S_t avec $t \in T$, l'ensemble d'indices étant totalement ordonné. Cet ensemble d'indices définit l'*horloge* du signal. Pour un signal S nous notons S.H son horloge. Les éléments de S.H sont les instants pour lesquels S est *présent*. Il est *absent* aux autres instants. Deux signaux ayant la même horloge sont dits *synchrones*. Certains signaux ne portent pas d'autre information que leur présence. En SIGNAL ces signaux sont du type event.

Les programmes synchrones imposent des relations entre horloges. Un programme SIGNAL, appelé aussi process, est un système d'équations sur les signaux qui exprime ces relations.

Le langage SIGNAL offre plusieurs niveaux de structuration (Besnard *et al.*, 2005). Nous utiliserons essentiellement les structures du *langage noyau*. Le langage noyau contient un ensemble réduit d'opérateurs sur les signaux de type boolean et event sur lesquels on calcule les structures temporelles du programme.

4.2. *Opérateurs de base*

Notons S_τ le statut de présence du signal S à un instant arbitraire τ. $\perp$ dénote l'absence du signal. S_k représente S à l'instant k pour lequel il est présent.

Pour chaque opérateur nous donnons sa signification et les contraintes qu'il impose sur les horloges et les ordres d'évaluation (causalité). Ces informations sont à la base du "calcul d'horloge". Pour représenter la dépendance causale nous utilisons la notation suivante : X $\rightarrow$ Y when B qui signifie Y dépend de X quand X et Y sont présents et B est true. X $\rightarrow$ Y est une notation abrégée pour X $\rightarrow$ Y when true.

Opérateurs mono-horloge

– *Opérateurs usuels*. Comme en LUSTRE on peut appliquer des opérateurs aux signaux (donc à des séquences). "Z := X op Y" impose que X, Y et Z aient la même horloge, on a alors $Z_k = op(X_k, Y_k)$. Les contraintes sont Z.H = X.H = Y.H, X $\rightarrow$ Z et Y $\rightarrow$ Z.

– *Retard*. "Z := X$1" impose lui aussi les mêmes horloges. Sa signification est $Z_k = X_{k-1}$. Il n'y a pas de problème de causalité puisque la relation porte sur des instants différents. La seule contrainte concerne les horloges Z.H = X.H.

Opérateurs multi-horloges

– *Sous-échantillonnage*. "Z := X **when** B" avec B booléen. Lorsque B_τ est true, $Z_\tau = X_\tau$, sinon $Z_\tau = \perp$. L'horloge de Z est une sous-horloge de celle de X : Z.H = X.H when B. Pour les causalités, B $\rightarrow$ Z et X $\rightarrow$ Z when B, c'est dire que le statut de B doit être connu avant celui de Z et lorque B est true le statut de X doit être connu avant celui de Z.

– *Mélange déterministe*. "Z := X **default** Y" fait des unions d'horloges : Z.H = X.H $\cup$ Y.H. Si X est présent, Z prend sa valeur, sinon si Y est présent Z prend la valeur de Y, sinon Z est absent. Ceci s'écrit $Z_\tau = X_\tau$ when $X_\tau \neq \perp$ et $Z_\tau = Y_\tau$ autrement. Les dépendances sont (X $\rightarrow$ Z) et (Y $\rightarrow$ Z when (H.Y - H.X)), c'est à dire pour les instants où Y est présent mais pas X.

4.3. *Programmation d'un compteur modulo*

Nous reprenons l'exemple du compteur précédemment programmé en LUSTRE. SIGNAL va permettre une description plus fine du comportement. Le programme SIGNAL est présenté progressivement avec des commentaires dans le programme (entre une paire de %) et hors programme.

```
process CptMod =
  { integer Mod; }  % paramètres %
  (                        % interface %
    ? event RAZ; integer Inc;   % entrées %
    ! integer Cpt; event Retenue; % sorties %
  )
  % placer ici le corps (équations) %
end;
```

Notons que dans ce programme l'entrée RAZ est du type event et non pas un booléen comme en LUSTRE. Le signal de sortie Retenue est aussi de ce type. Typiquement ce signal sera plus souvent absent que présent.

```
1  % corps %
2  (| Cpt ^= RAZ ^+ Inc
3   | LCpt ^= Inc
4   | LCpt := ( ZCpt when ^Inc ) + Inc
5   | ZCpt := Cpt$ init 0
6   | Cpt := 0 when RAZ default LCpt modulo Mod
7   | Retenue := when ( LCpt >= Mod)
8  |)
9  where
10   integer ZCpt, LCpt
```

Le corps du process peut utiliser des signaux locaux. Ils sont déclarés avec leur type à la ligne 10. Les équations sont définies des lignes 2 à 7. Le symbole '|' indique la *composition parallèle*. Le caractère déclaratif du langage fait que l'ordre des instructions est sans importance. Nous avons choisi de donner en premier des contraintes explicites sur des horloges. La ligne 2 impose une *synchronisation* : l'horloge de Cpt est l'union (^+) des horloges de RAZ et Inc. Ainsi le signal de sortie Cpt sera présent chaque fois que les signaux d'entrée RAZ ou Inc seront présents. Il sera absent autrement. La ligne 3 spécifie que le signal local LCpt est synchrone avec le signal d'entrée Inc. Bien noter que RAZ n'a pas d'influence directe sur LCpt. Les horloges des autres signaux sont définies implicitement, elles sont déterminées par le calcul d'horloge. Sur cet exemple très simple on peut aisément déduire de la ligne 5 que ZCpt et Cpt sont synchrones.

Les équations des lignes 4 à 6 concernent les valeurs des signaux. L'opérateur d'addition entière à la ligne 4 nécessite que tous les opérandes soient sur la même horloge. C'est le cas pour LCpt et Inc. La formulation intuitive : "LCpt := ZCpt + Inc" n'aurait pas convenu puisque ZCpt est sur une horloge différente des deux autres. En

revanche, le terme “ZCpt **when** ^Inc” est bien sur la même horloge. En effet, “^Inc” extrait l’horloge du signal Inc, puisque ZCpt.H $\supset$ Inc.H , “ZCpt **when** ^Inc” retourne un signal sur l’horloge de Inc.

Retenue est un event. Il ne doit être présent que lorsqu’il y a eu incrémentation et que le résultat était égal ou supérieur à Mod. “Retenue := **when** (LCpt >= Mod)” aurait pu être écrit “Retenue := **true** **when** (LCpt >= Mod)”. Retenue.H $\subset$ LCpt.H = Inc.H.

4.4. *Programmation complète du Comptage*

Cette solution prend pleinement en compte le caractère événementiel des signaux d’entrée (RAZ, Top, Avance). Et les sorties (S et M) ne sont effectivement présentes qu’en cas de mise à jour.

```
process Comptage =
{integer Modulus }
(
  ? event Top, RAZ, Avance;
  ! integer S, M;
)

(|  S ^= LTop ^+ RAZ
 |  M ^= RAZ ^+ LAvance ^+ Retenue
 |  H ^= RAZ ^+ Top ^+ Avance
 |  Mode := GestionMode(RAZ,H)
 |  LTop ^= Top when Mode
 |  LAvance ^= Avance when Mode
 |  IncS ^= LTop
 |  IncS := 1
 |  (S,Retenue) := CptMod {Modulus} (RAZ,IncS)
 |  IncM ^= LAvance ^+ Retenue
 |  IncM := (1 when LAvance default 0) + (1 when Retenue default 0)
 |  (M,RetenueM) := CptMod {Modulus} (RAZ,IncM)
|)

where
  event Retenue,RetenueM, LTop, LAvance, H;
  integer IncS, IncM;
  boolean Mode;

   process CptMod =
  { integer Modulus; }
  (
    ? event RAZ; integer Inc;
    ! integer Cpt; event Retenue;
  )
  (| Cpt ^= RAZ ^+ Inc
   | LCpt ^= Inc
```

```
35      | LCpt := (ZCpt when ^Inc) + Inc
36      | Cpt := 0 when RAZ
37        default LCpt modulo Modulus
38      | ZCpt := Cpt$ init 0
39      | Retenue := when (LCpt >= Modulus)
40    |)
41    where
42      integer ZCpt, LCpt
43    end % CptMod %;
44
45  process GestionMode =
46    (
47     ? event R,H;
48     ! boolean M;
49    )
50    (| B ^= H ^+ R
51     | B := R default false
52     | ZM := M$
53     | M := (true when B) default (B or ZM)
54    |)
55  where
56    boolean ZM init false,
57      B;
58  end; % GestionMode %
59end % Comptage %;
```

Le **process** Comptage réutilise le **process** CptMod défini précédemment (lignes 22 à 38). Il utilise également 4 signaux locaux (lignes 19 et 20).

Dans le corps, les lignes 8 à 11 définissent les horloges des signaux de sortie ‘S’ et ‘M’ ainsi que celle des signaux locaux ‘IncS’ et ÌncM’. ‘S’ ne sera présent qu’en mode normal et lorsque ‘Top’ ou ‘RAZ’ sont présents (ligne 8). L’horloge de M’ est définie de façon analogue à celle de ‘S’, mais en remplaçant l’horloge ‘Top’ par l’union des horloges ‘RetenueS’ et ‘Avance’ (ligne 9).

Simulation

On prend le modulo égal à 4 (au lieu de 60) pour faciliter l’observation des résultats.

Instant	1	2	3	4	5	6	7	8	9	10	11	12	13
Top	t	⊥	t	t	⊥	t	t	t	t	t	⊥	t	t
RAZ	⊥	⊥	⊥	⊥	t	⊥	⊥	⊥	⊥	⊥	⊥	t	⊥
Avance	⊥	t	⊥	t	t	⊥	⊥	⊥	t	⊥	t	t	t
S	⊥	⊥	⊥	⊥	0	1	2	3	0	1	⊥	0	1
M	⊥	⊥	⊥	⊥	0	⊥	⊥	⊥	2	⊥	3	0	1

5. ESTEREL

5.1. *Un langage synchrone impératif*

ESTEREL (Boussinot *et al.*, 1991) est un langage *synchrone impératif* spécialement adapté à l'expression du contrôle et des comportements réactifs. SYNCCHARTS (André, 1996) est un formalisme graphique qui a adopté la même sémantique qu'ESTEREL. Il est d'ailleurs intégré dans la version commerciale d'ESTEREL comme interface graphique du langage, sous le nom de *Safe State Machine*.

Pour comprendre le comportement d'un programme ESTEREL (ou d'un syncChart) il est commode de considérer que ce programme décrit les évolutions de *threads* concurrents et imbriqués. Les évolutions des threads sont synchronisées sur une *horloge globale*. A chaque *instant* de cette horloge, c'est à dire à chaque *réaction*, des threads peuvent être détruits, d'autres créés. Les threads qui existaient à l'instant précédent, reprennent leur exécution au point où ils s'étaient bloqués. Ceux qui sont créés prennent leur exécution depuis leur début. Durant une réaction un thread exécute les instructions spécifiées par le code (caractère impératif du langage). L'exécution d'un thread dans l'instant s'arrête sur des instructions particulières qui sont des points de contrôle indiqués explicitement par l'instruction "**pause**" ou par des pauses implicites liés à des structures de contrôle. L'exécution peut aussi s'arrêter parce que le thread termine (terminaison normale) ou bien parce qu'il est "tué". Ce dernier cas correspond aux *préemptions*. Les préemptions sont liées à la notion d'état hiérarchique qui est implicite en ESTEREL et explicite en SYNCCHARTS. Ainsi, il peut être plus aisé de comprendre les préemptions en SYNCCHARTS, c'est le point de vue adopté dans le papier (André, 2004). Le principe est qu'un thread peut être dans une région "préemptible" (dans les Activités d'UML on parlerait plutôt de région interruptible). La préemption devient effective lorsqu'une condition particulière est satisfaite ; tous les threads inclus dans la régions sont alors préemptés. Celà suppose que les conditions soient *évaluées à chaque réaction.*

Les communications entre les threads se font *uniquement* au travers de *signaux*, à l'exclusion d'autres moyens comme le partage de variables. La visibilité d'un signal peut être confinée à l'intérieur d'un bloc (**signal** ... **in** ... **end**) en ESTEREL ou d'un macro-état en SYNCCHARTS. A chaque instant, tous les threads qui se trouvent dans le domaine de visibilité d'un signal, peuvent accéder au statut de ce signal. Tout se passe comme si ce statut était diffusé instantanément. Comme nous l'avons expliqué dans les généralités de l'approche synchrone, ce statut est unique dans un instant. En ESTEREL un signal à un statut de présence. Celui-ci est réévalué à chaque instant (caractère événementiel). Les signaux dits *purs* n'ont pas d'autre attribut. Les signaux *valués* possèdent en plus une *valeur* typée. Contrairement à la présence, la valeur a un caractère rémanent : elle est conservée d'un instant à l'autre jusqu'à ce qu'une nouvelle valeur soit affectée par une instruction d'émission du signal.

5.2. *Instructions et constructions de base*

Le langage disponible dans la distribution commerciale est ESTEREL-v7. Par rapport aux versions précédentes, il offre des facilités accrues de structuration des programmes par les *units* (**data**, **interface**, **module**), de manipulations de données avec les *arrays* et de nouveaux types prédéfinis, ainsi que la possibilité nouvelle d'inclure des équations à la LUSTRE.

Pour illustrer le style de programmation propre à ESTEREL nous décrivons quelques instructions typiques.

– "**await** A; **emit** ARecu". **await** a un caractère fortement *asynchrone*. Le contrôle se bloque sur cette instruction (**pause** implicite) jusqu'à ce que le signal 'A' devienne présent. Le ';' qui suit signifie que lorsque l'instruction qui précède termine, le contrôle *passe instantanément en séquence*. **emit** ARecu est une *instruction instantanée* : elle termine dans l'instant même où elle a commencé et son effet est de rendre présent le signal 'ARecu'. Pour un signal valué, la nouvelle valeur du signal est donnée par une expression passée en paramètre.

– "{**await** A || **await** B}; **emit** ABRecus". L'opérateur || exprime la composition parallèle. Le bloc parallèle (ici entre les accolades) est terminé quand les deux branches sont terminées.

– "**loop** p **end loop**". Le corps 'p' de la boucle est répété indéfiniment. Il est interdit que 'p' soit une instruction instantanée.

– "**abort** p **when** S". Il s'agit d'une préemption. Lorsque 'p' termine, l'ensemble termine. Mais si 'p' est actif, la présence de 'S' fait avorter le tout.

5.3. *Programmation de Compteur*

La première version est purement impérative. C'est celle utilisée avec les versions antérieures à la V7.

```
1  module CompteurSecondes:
2  input RAZ, Top; % signaux purs
3  output CptS:integer, Retenue;
4  constant M=60:integer; % modulo
5  every RAZ do  % prioritaire sur Top
6    emit CptS(0);
7    every Top do
8      emit CptS((pre(?CptS)+1) mod M);
9      if ?CptS = 0 then emit Retenue end if
10   end every
11 end every
12 end module
```

"**every** RAZ **do** ... **end every**" est une boucle infinie avec préemption et réinitialisation immédiate lorsque 'RAZ' est présent. Le "**every** Top **do** ... **end every**"

qui est dans le corps du **every** précédent n'est pas exécuté en présence de 'RAZ'. Ainsi 'RAZ' a bien priorité sur 'Top'.

A la ligne 9, "?CptS" permet d'accéder à la valeur courante du signal 'CptS'. A la ligne 8, "**pre**(?CptS)" est un accès à la valeur de 'CptS' *avant* la réaction. Noter qu'au cours d'une réaction avec 'Top' présent et 'RAZ' absent, le respect de la règle de causalité impose que 'CptS' ait son état défini (ligne 8) *avant* son utilisation (ligne 9).

Un programme équivalent du point de vue comportemental écrit en ESTEREL-V7 est donné ci-dessous.

```
1  module CompteurSecondes:
2  input RAZ, Top; // signaux purs
3  output CptS:unsigned, Retenue;
4  constant M:unsigned=60; // modulo
5  every RAZ do
6    emit ?CptS <= 0; pause;
7    sustain {
8      Retenue if Top and (?CptS = 0),
9      ?CptS <= (pre(?CptS)+1) mod M if Top
10   }
11 end every
12 end module
```

Ce programme utilise le **sustain** qui était déjà présent dans les versions précédentes du langage. **sustain** S est équivalent à **loop emit** S **each tick** où **tick** est un signal prédéfini qui est présent à tout instant. Le **sustain** répète donc, sans fin, l'émission d'un signal. En ESTEREL-V7, il est possible de grouper des instructions entre une paire d'accolades. On peut ainsi écrire **sustain** {S,T} forme abrégée de **loop emit** S || **emit** T **each tick**.

Une nouveauté introduite dans la version V7 est la possibilité d'insérer des *équations*. Elles sont évaluées instantanément et leur ordre est sans importance (le compilateur détermine lui-même l'ordre des calculs). Ainsi les lignes 8 et 9 sont des équations qui définissent l'état des signaux 'Retenue' et 'CptS'. Le **sustain** fait que ces équations sont évaluées à chaque instant. Notons que la syntaxe des émissions a changé (ligne 6 par exemple).

5.4. *Programmation complète du Comptage en* ESTEREL *v5*

```
1  module COMPTAGE:
2  input RAZ, Top, Avance;
3  output S:integer, M:integer;
4  constant Modulus=4:integer; % instead of 60, for simulation purpose
5
6  signal
7    RetenueS,
```

```
8    IncrM: combine integer with +
9  in
10   every RAZ do
11     emit S(0); emit M(0);
12     [
13       every Top do
14         run IncrementerSeconde [constant Modulus/Mod]
15       end every
16     ||
17       every Avance do emit IncrM(1) end every
18     ||
19       every RetenueS do emit IncrM(1) end every
20     ||
21       every [Avance or RetenueS] do
22         run IncrementerMinute [constant Modulus/Mod]
23       end every
24     ]
25   end every
26 end signal
27 end module
28
29 module IncrementerSeconde:
30 output S:integer;
31 output RetenueS;
32 constant Mod:integer;
33
34 signal LS:integer in
35   emit LS(pre(?S)+1);
36   if ?LS >= Mod then
37     emit S(?LS - Mod);
38     emit RetenueS
39   else
40     emit S(?LS)
41   end if
42 end signal
43 end module
44
45 module IncrementerMinute:
46 output M:integer;
47 input IncrM:integer;
48 constant Mod:integer;
49
50 emit M((pre(?M)+?IncrM)mod Mod)
51
52 end module
```

Une trace d'exécution de ce programme est donnée dans le tableau ci-dessous. + note la présence d'un signal pur, - son absence. Pour les signaux valués la valeur est

suivi d'un exposant + lorsque cette valeur est émise, - en cas de non-émission mais persistance de la valeur. $\perp$ est une valeur non définie.

Instant	1	2	3	4	5	6	7	8	9	10
RAZ	-	-	-	-	-	+	-	-	-	-
Top	-	+	-	+	+	-	-	+	+	+
Avance	-	-	+	-	+	+	-	-	-	-
S	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	0^+	0^-	1^+	2^+	3^+
M	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	0^+	0^-	0^-	0^-	0^-

Instant	11	12	13	14	15	16	17	18	19	20
RAZ	-	-	-	-	-	-	-	+	-	-
Top	-	+	-	+	-	-	-	+	+	-
Avance	-	+	-	-	-	+	-	+	+	-
S	3^-	0^+	0^-	1^+	1^-	1^-	1^-	0^+	1^+	1^-
M	0^-	2^+	2^-	2^-	2^-	3^+	3^-	0^+	1^+	1^-

Ce programme comprend 3 modules. Ce n'est pas une nécessité. Nous avons choisi cette décomposition pour présenter les aspects modulaires du langage. Le module principal est "COMPTAGE" qui utilise les deux autres (instructions **run** ...).

"COMPTAGE" déclare deux signaux locaux (lignes 7 et 8) : "RetenueS" et "IncrM" qui sont utilisés pour les communications entre modules. Le signal 'incrM' est valué par les entiers mais il a un comportement particulier en cas d'émissions multiples dans un instant. Par défaut les signaux valués sont dits *single* : ils ne peuvent être émis qu'une fois au plus par instant. Il existe une deuxième catégorie de signaux appelés *combined* qui autorisent les émissions multiples. En cas d'émissions multiples la valeur prise par le signal est la combinaison des différentes valeurs émises dans l'instant. La fonction de combinaison doit être associative et commutative. Pour le signal 'IncrM' c'est l'addition entière qui a été retenue. Il peut être émis simultanément (lignes 17 et 19), en cas de présence simultanée de "Avance" et "RetenueS". A un instant d'émission, "IncrM" peut porter soit la valeur 1 (cas d'une seule émission), soit la valeur 2 (cas d'émissions doubles).

La gestion du Mode est très simple ici. Puisque le système ne fait rien tant que RAZ ne s'est pas produit, il suffit d'attendre la première occurrence de ce signal. C'est ce que fait le **every** RAZ **do** (ligne 10).

A la ligne 11, 'S' et 'M' sont émis avec la valeur 0 (conséquence de la présence de 'RAZ').

La réutilisation d'un module (ligne 14 et 22) se fait en instanciant un module (**run**) et en "branchant" les différents signaux et autres éléments d'interface. La syntaxe (incomplète) en utilisant une notation BNF [6] est :

6. { ... } est la répétition, [...] est optionnel, les symboles terminaux sont entre ' et ' (par exemple '['), les terminaux alphabétiques sont notés en fonte grasse, les non-terminaux sont écrits en italiques et définis ailleurs.

run [*instance-id* '/'] *module-id* '['
[[signal] {*new-name* '/' *old-name* ',' } ';']
[constant {*new-name* '/' *old-name* ',' } ';']
]

Tous les signaux branchés doivent être définis dans le contexte d'utilisation et avoir le bon type. L'omission du branchement explicite d'une signal S est interprétée comme S/S. C'est le cas de tous les signaux dans l'utilisation des deux modules de l'exemple. Seules les constantes ont été renommées (lignes 14 et 22).

5.5. *Programmation complète du Comptage en* ESTEREL *v7*

Le même programme a été écrit en ESTEREL-v7, avec un seul module.

```
1  data COMPTEUR_CFG:
2  constant N:unsigned<>=4; // au lieu de 60, pour simulation
3  type CPTModN = unsigned<N>;
4  end data
5
6  main module Comptage:
7  extends COMPTEUR_CFG;
8  input RAZ, Top, AvanceRapide;
9  output S:CPTModN, M:CPTModN;
10
11 signal
12   Sa0,
13   incrM:temp unsigned<3> combine + // 0..2
14 in
15   // Mode Oisif : attendre la 1ere occurrence de RAZ
16   every RAZ do  // remise a 0 prioritaire
17     emit {
18       ?S <= 0,
19       ?M <= 0
20     };
21     pause; // incrementations possibles
22     sustain {
23       Sa0 if S and (?S = 0), // present quand S repasse a 0
24       ?incrM <= 1 if Sa0, // propagation de la retenue S
25       ?incrM <= 1 if AvanceRapide, // prise en compte AvanceRapide
26       ?S <= (pre(?S)+1) mod N if Top, // incrementation S
27       ?M <= (pre(?M)+?incrM) mod N if incrM // incrementation M
28     }
29   end every
30 end signal
31 end module
```

Les simulations peuvent être visualisées sous forme de chronogramme (Figure 6).

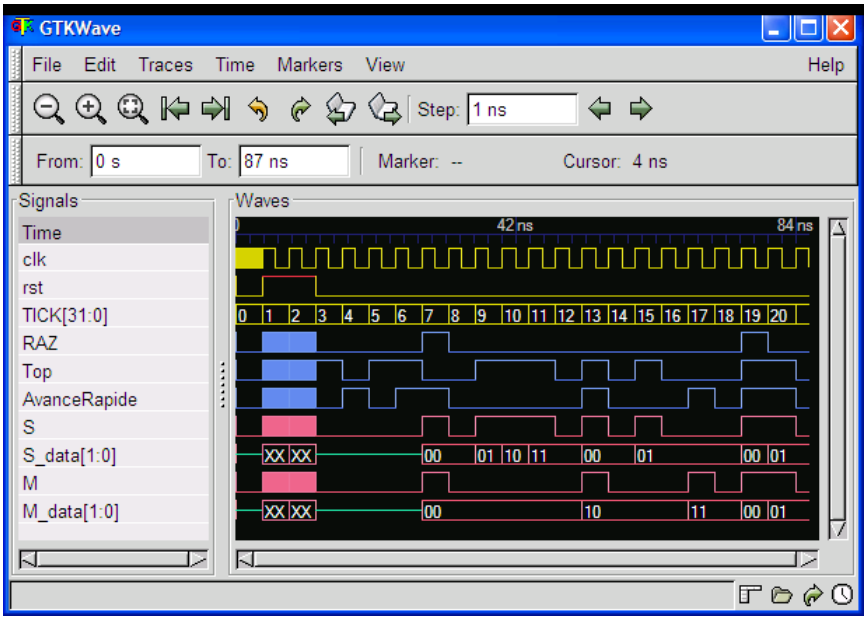

Figure 6. *Chronogrammes d'execution de Comptage (Esterel).*

Parmi les nouveautés introduites, notons l'existence d'*unités* de programmation qui peuvent être

– des unités de données (**data** *unit-id* : . . . **end data**)

– des unités d'interface (**interface** *unit-id* : . . . **end interface**)

– des modules (**module** *unit-id* : . . . **end module**)

Les unités sont réutilisés (ligne 7) par l'instruction **extends** *unit-id.* Les signaux importés peuvent être raffinés (instruction **refine** *signal-id* ...).

Les unités de données peuvent définir des types génériques, ce qui permet de paramétrer facilement les programmes.

Les équations (lignes 23 à 27) expriment de façon concise et lisible les différentes actions conditionnelles à réaliser en mode "Normal".

6. Conclusions et Perspectives

Les langages synchrones sont des langages spécialisés dans la programmation de systèmes réactifs. Ils concilient déterminisme avec évolutions concurrentes et communication. Leur style de programmation peut être déclaratif ou impératif. Le premier se prête bien aux traitements flots de données, le second aux applications à contrôle prépondérant. Ces langages tirent parti de leurs fondements mathématiques pour offrir

des possibilités étendues d'optimisation de code et de vérification formelle de propriétés. Ces avantages sont pleinement exploités dans les systèmes tels que celui qui a servi à illustrer cette présentation. Dans ce système, qui est aisément réalisé comme un contrôleur centralisé, la notion de *temps global* est tout à fait réaliste. Les applications réactives avec temps global étaient les cibles initiales des langages synchrones. Cet article a essentiellement développé ce point de vue. Les langages synchrones sont maintenant beaucoup largement utilisés, par exemple dans les systèmes embarqués.

Une première difficulté liée à la considération d'un temps logique résulte de la durée physique des actions. Un langage synchrone peut appeler des traitements dans un langage hôte. Le caractère atomique des réactions synchrones fait que celles-ci peuvent avoir des durées importantes, incompatibles avec un fonctionnement sous contraintes de temps réel. Pour résoudre ces problèmes, les langages synchrones utilisent une technique de raffinement du temps. Partant des ordonnancements calculés par les compilateurs, des threads d'exécution, à durée d'exécution prévisibles, sont déduits. Ils sont ensuite décomposés en séquence de "microthreads" exécutés dans des instants "plus fins". Cette technique est utilisée dans des outils comme SYNDEX (`http://www.syndex.org`).

Une deuxième difficulté est liée à la référence à une horloge globale. C'est le cas pour les applications réparties (sites distants, mais aussi systèmes multi-processeurs et même systèmes multi-horloges tels qu'on les trouvent dans les circuits complexes). SIGNAL supporte nativement les aspects multi-horloges, ESTEREL prévoit l'extension du langage dans cette direction. Des travaux théoriques et techniques ont permis le déploiement de programmes synchrones sur des architectures GALS (Globally Asynchronous Locally Synchronous) (Benveniste *et al.,* 1999) et des réseaux de processeurs (Caspi *et al.,* 1999).

Une voie très prometteuse pour les approches synchrones est leur utilisation dans la conception à base de composants. Un langage comme SIGNAL permet une spécification précise du comportement des composants avec possibilités d'abstraction et de raffinements.

7. Bibliographie

André C., « Representation and Analysis of Reactive Behaviors : A Synchronous Approach », *Computational Engineering in Systems Applications (CESA)*, IEEE-SMC, Lille (F), p. 19-29, July, 1996.

André C., « Computing SyncCharts Reactions », *Electronic Notes in Theoretical Computer Science*, vol. 88, p. 3-19, October, 2004.

Benveniste A., Berry G., « The Synchronous Approach to Reactive and Real-Time Systems », *Proceeding of the IEEE*, vol. 79, n° 9, p. 1270-1282, September, 1991.

Benveniste A., Caillaud B., Le Guernic P., « From Synchrony to Asynchrony », *CONCUR'99 : Concurrency Theory*, vol. 1664, Heidelberg (D), p. 162-177, March, 1999.

Benveniste A., Caspi P., Edwards S. A., Halbwachs N., Le Guernic P., de Simone R., « The Synchronous Languages 12 Years Later », *Proceedings of the IEEE*, vol. 91, n° 1, p. 64-83, 2003.

Benveniste A., Caspi P., Le Guernic P., Halbwachs N., Data-Flow Synchronous Languages, Technical Report n° 2089, INRIA, IRISA, Rennes, (F), October, 1993.

Berry G., The Constructive Semantics of PURE ESTEREL, Technical Report n° electronic version, Esterel Technologies, July, 1999.

Berry G., « The Foundations of Esterel », *in* , C. S. G. Plotkin, , M. Tofte (eds), *Proof, Language and Interaction : Essays in Honour of Robin Milner*, MIT Press, 2000.

Besnard L., Gautier T., Le Guernic P., *SIGNAL V4 – INRIA version : ReferenceManual*, IRISA/CNRS, Campus de Beaulieu, 35042 Rennes Cedex (France). March, 2005.

Boussinot F., De Simone R., « The ESTEREL Language », *Proceeding of the IEEE*, vol. 79, n° 9, p. 1293-1304, September, 1991.

Boussinot F., SugarCubes Implementation of Causality, Technical Report n° 3487, INRIA, September, 1998.

Caspi P., Girault A., Pilaud D., « Automatic distribution of reactive systems for asynchronous networks of processors », *IEEE Transactions on Software Engineering*, vol. 25, p. 416-427, March, 1999.

Caspi P., Pouzet M., « A co-iterative characterization of synchronous stream functions », *Proceedings 1st Workshop on Coalgebraic Methods in Computer Science*, vol. 11, Lisbon (Portugal), March, 1998. http ://www.sciencedirect.com.

Edwards S. A., *Languages for Digital Embedded Systems*, Kluwer Academic Publishers, Boston(MA), 2000.

Halbwachs N., Caspi P., Raymond P., Pilaud D., « The Synchronous Data Flow Programming Language LUSTRE », *Proceeding of the IEEE*, vol. 79, n° 9, p. 1305-1320, September, 1991.

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

Harel D., Rumpe B., Modeling Languages : Syntax, Semantics, and All That Stuff. Part I : The Basic Stuff, Technical report, Weizmann Institute Of Science – Mathematics & Computer Science, Rehovot (Israel), October, 2000. Technical Report MCS00-16.

Harel D., Rumpe B., « Meaningful Modeling : What's the Semantics of 'Semantics' ? », *Computer*, vol. 37, n° 10, p. 64-72, 2004.

Harel D., « STATECHARTS : A visual formalism for complex systems », *Science of computer programming*, vol. 8, p. 231-274, 1987.

Le Guernic P., Gautier T., Le Borgne M., Le Maire C., « Programming Real-Time Applications with SIGNAL », *Proceeding of the IEEE*, vol. 79, n° 9, p. 1321-1336, September, 1991.

Le Guernic P., Talpin J.-P., Le Lann J.-C., Polychrony for system design, Technical report, IRISA, Rennes (F), February, 2003. Technical Report 4715.