Lucid Synchrone, un langage de programmation de systèmes réactifs *

Paul Caspi †
VERIMAG

Grégoire Hamon
Chalmers University

Marc Pouzet ‡
LRI, Univ. Paris-Sud 11

1 Introduction

Ce chapitre présente Lucid Synchrone , un langage dédié à la programmation de systèmes réactifs. Il est fondé sur le modèle synchrone de Lustre [29] qu'il étend avec des caractéristiques présentes dans les langages fonctionnels tels que l'ordre supérieur ou l'inférence des types. Il offre un mécanisme de synthèse automatique des horloges et permet de décrire, dans un cadre unifié, une programmation flot de données et une programmation par automates.

Ce chapitre est une présentation du langage, destinée à la fois au programmeur cherchant à se familiariser avec celui-ci, mais aussi au chercheur s'intéressant aux systèmes réactifs et aux techniques qui peuvent être appliquées au niveau d'un langage, pour faciliter les activités de spécification, de programmation et de vérification.

1.1 Langages pour les systèmes réactifs

Nous nous intéressons ici aux langages de programmation dédiés à la réalisation de systèmes réactifs [36]. Ces systèmes interagissent continuellement avec leur environnement extérieur et sont soumis à des contraintes temporelles fortes: gestion du freinage d'urgence, système de régulation, pilote automatique dans un avion, chronomètre électronique, etc. Cette informatique, le plus souvent embarquée dans des processeurs ou calculateurs dédiés, touche aujourd'hui tous les domaines. La question de la réalisation de ces systèmes par des moyens informatiques se pose dès le milieu des années 70 quand s'amorce le passage de systèmes mécaniques ou électroniques à des systèmes logiques. Les premières implantations utilisent alors les langages informatiques du moment tels que l'assembleur ou des langages généralistes (par exemple C ou Ada). La prépondérance de langages de bas niveau s'explique par les impératifs de contrôle fin des ressources inhérents à ces systèmes. Ils s'exécutent le plus souvent sur du matériel ayant des capacités de mémoire limitées et le temps de réaction doit pouvoir être calculé statiquement. Dans ces applications embarquées, nombre d'entre elles concernent progressivement des fonctions critiques (commande de vol logicielle, systèmes de surveillance nucléaire). À ce titre, elles sont alors soumises aux contraintes imposées par les autorités de certification indépendantes qui doivent évaluer la qualité du logiciel avant la mise en service (norme DO-178B pour l'avionique ou IEC-61508 pour les systèmes critiques).

Les premiers langages utilisés montrent assez vite leurs limites. La description de bas niveau qu'ils autorisent est souvent très éloignée de la spécification des systèmes, rendant l'implantation source d'erreurs et les activités de certification délicates. Un système réactif étant concurrent par nature — le système évolue en parallèle avec son environnement et il se décompose en sous-systèmes évoluant en parallèle — il était donc normal de se tourner vers des modèles de programma-

*Chapitre 7 de l'ouvrage *Systèmes Temps-réel: Techniques de Description de Vérification - Théorie et Outils, volume 1*, pages 217-260. Editeur: Nicolas Navet. Hermes, 2006.

†Laboratoire VERIMAG, Centre Equation, 2, avenue de Vignate, 38610 GIERES, France. Email: Paul.Caspi@imag.fr

‡LRI, Université Paris-Sud 11, 91405 Orsay, France. Email: Marc.Pouzet@lri.fr

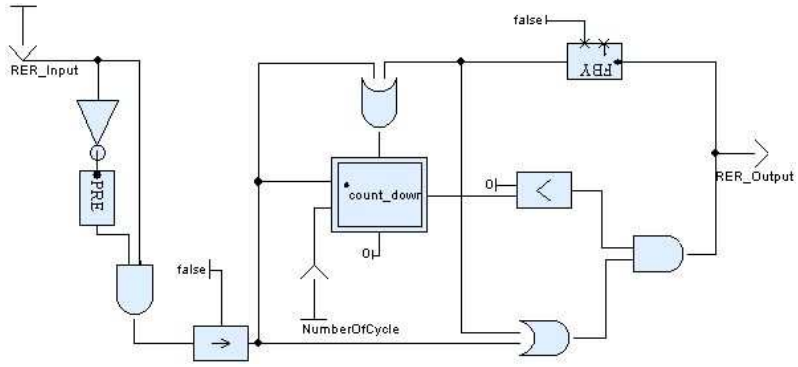


Figure 1: Un exemple de planche SCADE

tion concurrente. Les approches traditionnelles de la concurrence posent cependant d'importants problèmes d'analyse des systèmes, de non-déterminisme et de compilation et seront peu utilisées dans les domaines où la sécurité est importante. Ces approches étaient également éloignées des formalismes de l'ingénieur du domaine (systèmes échantillonnés, machines d'état fini).

1.1.1 Les langages synchrones

Les industries dites critiques (par exemple dans l'avionique ou l'énergie) sont les premières à rechercher et adopter d'autres techniques de développement pouvant répondre à leurs impératifs de sécurité. Une réponse est proposée au début des années 80 avec l'introduction des langages synchrones [28, 3] dont les plus connus sont Esterel [6], Lustre [29] et Signal [4]. Ces langages sont conçus spécifiquement pour décrire des systèmes réactifs. Cette et cette idée est essentielle: elle conduit à proposer des langages ayant un pouvoir expressif limité mais adapté à leur domaine d'application [5]. En contrepartie, ils disposent d'analyses de programme dédiées offrant des garanties à la compilation sur le comportement du système à l'exécution. Le caractère temps-réel (exécution en temps et mémoire bornée) ou l'absence de blocage, par exemple, sont garantis par construction. Ces langages se sont développés et ont connu des succès importants, participant ainsi à la diffusion de méthodes formelles dans l'industrie.

La base commune de ces langages est un modèle mathématique original de la concurrence et du temps, et une hypothèse de travail appelée l'*hypothèse synchrone*. L'hypothèse synchrone se fonde sur l'idée de détacher la description fonctionnelle d'un système des contraintes de l'architecture sur laquelle il est exécuté. La fonctionnalité du système peut être décrite en faisant une hypothèse de temps nul sur les calculs et les communications dès lors que l'on peut vérifier, *a posteriori*, que la machine est suffisamment rapide pour les contraintes imposées par l'environnement. C'est le modèle classique zéro délai de l'électronique ou de l'automatique qui permet, au moment de la spécification, de raisonner en temps logique sans tenir compte des temps réels des calculs. Sous cette hypothèse, le non-déterminisme dû au temps partagé disparaît puisque la machine est supposée disposer d'un nombre illimité de ressources. On combine ainsi la concurrence, c'est-à-dire la possibilité de décrire un système comme la composition parallèle de sous-systèmes, et le déterminisme. L'hypothèse synchrone permet de concevoir des langages de programmation de haut niveau offrant des constructions de concurrence mais pouvant être compilés vers du code séquentiel. On parvient ainsi à réconcilier les descriptions de haut niveau et l'efficacité d'une implantation séquentielle. En pratique, l'adéquation entre le temps logique et le temps physique est vérifiée en utilisant des outils d'analyse de temps de réaction dans le pire cas sur le code obtenu après compilation (voir le chapitre 5 du volume II de ce traité).

Le langage Lustre s'est fondé sur l'idée qu'une restriction synchrone d'un langage fonctionnel de suites, dans l'esprit de Lucid [2] et des réseaux de processus de Kahn [38], permettrait de

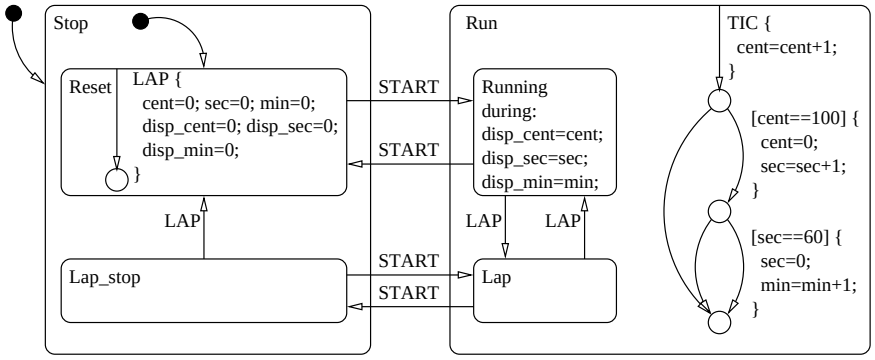


Figure 2: Un chronomètre en Stateflow.

programmer des applications réactives [1]. Il est basé sur un modèle de programmation flot de données permettant de spécifier le système par un ensemble d'équations de suites. Ces suites, ou flots, servent à représenter les communications entre les divers composants du système. Ce modèle de programmation correspond aux spécifications manipulées par les ingénieurs du domaine, permettant ainsi de réduire la distance entre la spécification et l'implantation. De plus, le langage se prête très bien à l'application de techniques de vérification (voir le chapitre 6 de ce traité), de test [53] et de compilation [30]. L'adéquation du langage avec les habitudes du domaine et l'efficacité des techniques de certification vont faire le succès de Lustre et de sa version industrielle graphique des équations Lustre proche des diagrammes utilisés par les ingénieurs en automatique et les concepteurs de circuit (voir figure 1). Il dispose d'outils de validation formelle et son générateur de code est qualifié (DO-178B niveau A) réduisant considérablement les phases de test. Aujourd'hui, SCADE est utilisé dans de nombreuses industries critiques et est au cœur du développement de l'informatique de l'Airbus A380.

1.1.2 Développement par modèle

Dans le même temps, les industries non critiques continuent à utiliser des langages généralistes ou dédiés de bas niveau (langages de la norme IEC-1131). Les impératifs de sécurité sont moindres, et ne justifient pas le coût d'un changement de méthode de développement. C'est poussé par une croissance galopante de la taille et de la complexité des systèmes et par des cycles de développement de plus en plus court qu'elles vont finalement se tourner vers de nouvelles méthodes. C'est l'adoption récente et massive d'environnements de développement par modèle, l'utilisation importante de Simulink/Stateflow et l'apparition d'UML. Ces environnements intégrés proposent de décrire les systèmes dans des formalismes graphiques de haut niveau proches des automates. Ils fournissent alors un ensemble d'outils pour la simulation, le test ou la génération de code. Ici encore, on a réduit la distance entre spécification et implantation, le système étant généré automatiquement à partir du modèle.

Ces outils n'ont généralement pas été conçus dans une optique de développement formel. Leur sémantique est partiellement spécifiée et est alors informelle. Ceci est un frein important à toute activité de vérification et à l'obtention, par des moyens automatiques d'un code certifié. C'est également un problème grandissant pour l'activité de développement elle-même: la complexité des systèmes grandissant, le risque d'erreur d'interprétation des modèles grandit également. Un exemple de tel environnement est Simulink/Stateflow [47] (voir figure 2), aujourd'hui standard dans de nombreuses industries. Simulink/Stateflow n'a pas de sémantique formelle et le langage offre de multiples manières d'écrire des programmes qui échoueront à l'exécution. En contrepartie, l'environnement offre une panoplie complète d'outils et de nombreuses librairies permettant de

[1] Lustre vient de Lucid Synchrone et Temps Réel.

modéliser à la fois le système, son environnement et son architecture, de les simuler, et de générer les codes correspondant au système.

1.1.3 Des besoins convergents

Aujourd'hui, les besoins des industries, critiques ou non, se rejoignent. Toutes sont confrontées à des problèmes de croissance des systèmes en taille et en complexité. Les langages synchrones, conçus pour répondre aux besoins d'un domaine d'application très précis (par exemple l'automatique pour Lustre) ne sont pas directement adaptés à la description de systèmes complexes issus de plusieurs domaines. Les besoins de vérification et d'analyse des systèmes, un temps cantonnés aux industries critiques, sont maintenant présents partout. La taille des systèmes impose une automatisation des tâches de vérification, elle-même rendue essentielle par la diffusion des systèmes et les coûts gigantesques liés à des erreurs de conception non détectées. Ces besoins convergents nécessitent notamment:

- des mécanismes d'abstraction, permettant de se rapprocher toujours plus de la spécification des systèmes, une meilleure réutilisation de composants et la création de librairies;
- des techniques d'analyse automatiques et modulaires et de génération de code permettant d'obtenir du code garantissant l'absence de certaines erreurs à l'exécution;
- un modèle de programmation permettant de combiner, au sein d'un même langage, des descriptions flot de données issues de modèles continus et des descriptions à contrôle prépondérant basées sur des automates.

1.2 Lucid Synchrone

Le langage Lucid Synchrone a été introduit pour répondre à des besoins d'extensions de Lustre avec des mécanismes de plus haut niveau que ceux dont il disposait jusque-là: besoin de mécanisme de synthèse des types et des horloges, modularité, mélange de traits impératifs et flot de données, etc. Il s'est fondé sur l'observation qu'il existait une grande proximité entre trois courants d'idées, la programmation synchrone flot de données et les outils de l'automaticien, la sémantique des réseaux de Kahn et l'évaluation paresseuse dans les langages fonctionnels [59]. En étudiant ces relations et en reformulant le modèle synchrone dans le cadre général de la théorie des langages fonctionnels typés, nous cherchions à répondre à des questions de deux ordres:

- des questions théoriques sur la sémantique, les analyses statiques et la compilation des langages de flots synchrones;
- des questions pratiques portant sur l'expressivité de Lustre et les possibilités d'extensions afin de répondre à des besoins exprimés par les utilisateurs de SCADE.

Les premiers travaux, d'ordre théorique, ont montré qu'il était possible de définir une extension fonctionnelle de Lustre permettant de combiner l'expressivité de la programmation fonctionnelle en flots paresseux avec l'efficacité due au synchronisme [15]. Le calcul d'horloge — une analyse de cohérence des différents rythmes du système — a été exprimé sous la forme d'un système de types dépendants en le généralisant à l'ordre supérieur [16]. Il restait alors à comprendre comment étendre les méthodes de compilation existantes. Nous avons été aidés, en cela, par l'utilisation de techniques venant des co-algèbres permettant de retrouver et de généraliser la manière traditionnelle de compiler les programmes SCADE [17]. Ces différents résultats ont servi de base aux premières implantations d'un compilateur (V1). Bien que théoriques, ces premiers résultats se sont révélés utiles en pratique. L'expression du calcul d'horloge sous forme d'un système de type permet d'inférer automatiquement les horloges, les rendant ainsi plus facilement utilisables par le programmeur. De même, la conception d'un système de types avec polymorphisme permet d'augmenter la réutilisation de code [19]. Ces mécanismes de synthèse automatique se révèlent indispensables dans un outil graphique tel que SCADE où les programmes sont avant tout dessinés. Ils existent, aussi, mais cachés, dans les outils issus de l'automatique tels que Simulink [14].

Une collaboration s'engage alors avec l'équipe de développement de SCADE chez Telelogic (et maintenant Esterel-Technologies) visant à écrire un nouveau compilateur de SCADE. Ce compilateur, appelé ReLuC (pour *Retargetable Lustre Compiler*) se fonde sur les premiers résultats théoriques mis en oeuvre dans Lucid Synchrone. Le calcul d'horloge par typage ainsi que certaines constructions y sont intégrés.

Les bases du langage étant posées, les travaux s'orientent d'abord vers la conception d'analyses modulaires de programmes: analyse des boucles de causalité, analyse d'initialisation [25, 21, 23] en les exprimant sous forme de systèmes de types. L'analyse d'initialisation est développée en collaboration avec Jean-Louis Colaço d'Esterel-Technologies et intégrée à la fois dans le compilateur de Lucid Synchrone et dans le compilateur ReLuC. Expérimentée sur des exemples de taille réelle (plus de 50 000 lignes de code), elle se révèle très rapide et permet de réduire le nombre de fausses alarmes. Le calcul d'horloge étant fondé sur un calcul de types dépendants, il était naturel de le plonger dans l'assistant de preuve Coq [24], permettant par là même d'établir une preuve de correction solide [9]. En observant les planches SCADE, nous réalisons que ce calcul peut être obtenu par un système plus simple à la ML et coïncidant avec le système de type de Laüfer & Odersky [40]. Bien que moins puissant, il se révèle suffisamment expressif en pratique et peut être implanté beaucoup plus efficacement [22]. Il est utilisé dans la version actuelle du compilateur (V3). Parallèlement à la conception d'analyses de programmes, plusieurs extensions du langage sont envisagées. La première concerne l'extension d'un noyau Lustre avec une primitive de réinitialisation modulaire [34], l'ajout des types sommes et une construction de filtrage [33]. Ces travaux révèlent tout l'intérêt du mécanisme des horloges des langages synchrones et introduits dès l'origine dans Lustre et Signal. Elles permettent de donner une sémantique simple et précise aux structures de contrôle qui peuvent être traduites dans le langage noyau avec horloges en récupérant ainsi le générateur de code existant. Ce travail se poursuit en collaboration avec Jean-Louis Colaço et Bruno Pagano d'Esterel-Technologies et aboutit à une proposition d'extension de Lustre avec des constructions d'automates [20] dans l'esprit des Automates de Modes de Maraninchi et Rémond [45]. Cette extension repose aussi largement sur le mécanisme des horloges et les automates sont traduits vers le langage noyau. Dans le cas de SCADE, cette approche permet de réutiliser le générateur de code qualifié existant, simplifiant ainsi les phases de certification. L'ajout d'automates a été implanté à la fois dans le compilateur de Lucid Synchrone (V3) et dans le compilateur ReLuC. Elle sera intégrée dans la prochaine version industrielle de SCADE.

Nous présentons maintenant le langage et les extensions considérées à travers des exemples (section 2). Cette présentation est volontairement peu technique et nous renvoyons le lecteur aux articles en références pour les détails de sémantique et de compilation. L'objectif ici est plutôt de montrer, de manière progressive, comment étendre un noyau synchrone flot de données à la Lustre avec de nouvelles constructions. Dans la section 3, nous discutons des travaux proches et concluons dans la section 4.

2 Lucid Synchrone

On ne peut parler du langage sans parler de ses deux parents que sont le langage OCaml [42] d'une part et le langage Lustre de l'autre. Le lecteur familier des langages fonctionnels à la ML peut le voir comme un sous-ensemble de OCaml manipulant des flots synchrones. Le lecteur familier de Lustre préférera sans doute le voir comme une extension de Lustre avec des traits fonctionnels et des structures de contrôle. Le langage ayant largement évolué depuis la première implantation de 1995, nous présentons ici la version actuelle (V3)[2].

2.1 Un Langage à la ML sur des flots

2.1.1 Les flots comme objets de base

Lucid Synchrone est un langage flots de données et, à ce titre, il ne permet de manipuler que des flots, ou suites infinies de valeurs. Ainsi, parler d'un objet x revient à parler de la suite infinie

[2]Le langage est accessible à l'adresse www.lri.fr/ pouzet/lucid-synchrone.

$x_0, x_1, x_2, ...$ des valeurs que x prend au cours du temps. De même, la constante `1` ne désigne pas la valeur scalaire entière 1, mais la suite infinie $1, 1, 1, ...$ Enfin, le type `int` désigne le type des flots d'entiers. Les fonctions scalaires classiques (par exemple, `+`, `*`) sont également étendues aux flots. Elles sont alors appliquées points à point à leurs arguments:

`c`	t	f	t	...
`x`	x_0	x_1	x_2	...
`y`	y_0	y_1	y_2	...
`x+y`	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	...
`if c then x else y`	x_0	y_1	x_2	...
`if c then (x,y) else (0,0)`	(x_0, y_0)	$(0, 0)$	(x_2, y_2)	...

`x` et `y` étant deux flots d'entiers, l'opération `x+y` produit un flot, obtenu en additionnant point à point les valeurs de `x` et `y`. Le caractère synchrone se retrouve aisément ici: au i-ème instant, tous les flots prennent leur i-ème valeur.

Il est possible de construire des tuples de flots et ces tuples peuvent encore être vus comme des flots. Cela permet d'avoir une vue homogène du système: chaque valeur manipulée est susceptible d'évoluer au cours du temps.

2.1.2 Opérations temporelles: décalage et initialisation

L'opération de décalage initialisé `fby` (ou *followed by*) est empruntée au langage Lucid [2], l'ancêtre des langages flot de données. Cette opération prend un premier argument qui va servir à donner une valeur initiale au résultat et un second argument qui est le flot à retarder. Dans l'exemple suivant, `x fby y` retourne le flot `y` retardé d'un instant et initialisé par la première valeur de `x`.

Il est souvent très commode de séparer délai et initialisation. Ceci est obtenu en utilisant les opérateurs de retard `pre` (pour *previous*) et d'initialisation `->`. `pre x` retarde son argument `x` et a une valeur non spécifiée *nil* au premier instant. `x -> y` retourne la première valeur de `x` au premier instant puis la valeur courante de `y`. L'expression `x -> pre y` est donc équivalente à `x fby y`.

`x`	x_0	x_1	x_2	...
`y`	y_0	y_1	y_2	...
`x fby y`	x_0	y_0	y_1	...
`pre x`	*nil*	x_0	x_1	...
`x -> y`	x_0	y_1	y_2	...

L'opérateur `pre` introduisant dans le langage la possibilité d'avoir des valeurs non définies (ici représentées par *nil*), il est primordial de vérifier que le comportement du programme ne dépend pas de ces valeurs. Ce sera le rôle de l'analyse d'initialisation.

2.2 Fonctions de flots

Le langage distingue deux classes de fonctions de flots : les fonctions combinatoires et les fonctions séquentielles. Une fonction est combinatoire lorsque sa sortie à l'instant courant dépend seulement de la valeur de son entrée à l'instant courant. Il s'agit donc d'une fonction sans mémoire. Une condition suffisante pour qu'une expression soit combinatoire est qu'elle ne contienne aucun délai, d'opération d'initialisation ou de construction d'automate. Cette condition suffisante se vérifie facilement au cours du typage.

Un additionneur 1-bit est un bon exemple de fonction combinatoire. Il est paramétré par trois entrées booléennes, des opérandes `a`, `b`, une retenue `c` et calcule le résultat `s` et la retenue suivante `co`.

```
let xor (a, b) = (a & not(b)) or (not(a) & b)

let full_add (a, b, c) = (s, co) where
       s = xor (xor (a, b), c)
   and co = (a & b) or (b & c) or (a & c)
```

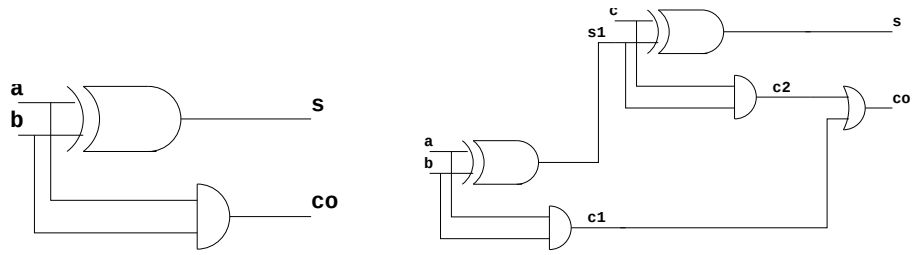


Figure 3: Définition hiérarchique d'un additionneur 1-bit

Lorsque ce texte est fourni au compilateur, on obtient la réponse:

```
val xor :  bool * bool -> bool
val xor :: 'a * 'a -> 'a
val full_add :  bool * bool * bool -> bool * bool
val full_add :: 'a * 'a * 'a -> 'a * 'a
```

Pour chaque déclaration, le compilateur indique les types (`:`) et les horloges (`::`) inférés automatiquement. La signature de type `bool * bool -> bool` indique que la fonction `xor` est une fonction combinatoire qui, à tout couple de flots booléens, retourne un flot booléen. La signature d'horloge `'a * 'a -> 'a` indique que `xor` est une fonction qui préserve les longueurs: elle renvoie une valeur pour chaque entrée. Nous reviendrons sur les horloges par la suite.

Un additionneur peut être décrit de manière plus efficace par la composition parallèle de deux demi-additionneurs. La représentation graphique du programme est donnée figure 3.

```
let half_add (a,b) = (s, co)
  where s = xor (a, b) and co = a & b

let full_add(a,b,c) = (s, co) where
  rec (s1, c1) = half_add(a,b)
  and (s, c2) = half_add(c, s1)
  and co = c2 or c1
```

Les fonctions *séquentielles* (ou à mémoire) sont telles que leurs sorties à l'instant n peut dépendre de l'histoire de leurs entrées, c'est-à-dire des valeurs des entrées aux instants k ($k \leq n$). Les fonctions séquentielles sont introduites par le mot clef `node` et elle reçoivent une signature de type différente des fonctions combinatoires. La fonction `edge` qui détecte un front montant s'écrira ainsi:

```
let node edge c = false -> c & not (pre c)

val edge :  bool => bool
val edge :: 'a -> 'a
```

Le diagramme suivant décrit une exécution possible.

`c`	f	f	t	t	f	t	...
`false`	f	f	f	f	f	f	...
`c & not (pre c)`	*nil*	f	t	f	f	t	...
`edge c`	f	f	t	f	f	t	...

La signature `bool => bool` indique que la fonction `edge` est une fonction transformant une suite booléenne en une suite booléenne et que son résultat dépend de l'histoire de ses entrées. On distingue donc ces deux classes au moyen de types. Le caractère combinatoire est vérifié durant la phase de typage. Ainsi, l'oubli du mot-clef `node` entraîne un rejet du programme.

```
let edge c = false -> c & not (pre c)

>let edge c = false -> c & not (pre c)
>             ^^^^^^^^^^^^^^^^^^^^^^^^
This expression should be combinatorial.
```

Dans un langage flot de données, les flots peuvent être définis de manière mutuellement récursive, les uns en fonction des autres et dans un ordre quelconque. Ceci permet de décrire un système comme une composition parallèle de sous-systèmes élémentaires. Ainsi, la fonction qui décompte le temps à partir d'un délai initial s'écrira:

```
let node count d t = ok where
  rec ok = (cpt = d)
  and cpt = d -> if pre cpt = 1 then d
                 else if t then pre cpt - 1 else pre cpt
val count : int -> int => bool
val count :: 'a -> 'a -> 'a
```

Ainsi, `count d t` retourne la valeur booléenne vraie lorsque `d` occurrences de `t` ont été reçues.

d	3	2	2	2	2	4	...
t	t	t	t	t	t	t	...
cpt	3	2	1	2	1	4	...
ok	t	f	f	t	f	t	...

Étant dans un langage fonctionnel, il est possible d'écrire des applications partielles en fixant l'un des paramètres de la fonction.

```
let count10 t = count 10

val count10 :  int => int
val count10 :: 'a -> 'a
```

2.3 Systèmes multi-horloges

Jusque-là, nous avons défini des systèmes où tous les processus mis en parallèle évoluent au même rythme. À chaque instant n, tous les flots du système prennent leur n-ième valeur. Ce sont des systèmes dits mono-horloges et un circuit synchrone classique en est un bon exemple.

Nous allons voir maintenant comment décrire et composer des systèmes évoluant à des rythmes différents. Ceci est fondé sur un mécanisme d'horloges logiques introduit à l'origine dans Lustre et Signal. L'horloge d'un flot est une information caractérisant les instants où le flot est présent. Certaines opérations vont permettre de produire un flot plus lent (échantillonnage) ou plus rapide (sur-échantillonnage). Dans Lucid Synchrone, l'information d'horloge est un type et elle est synthétisée automatiquement par le compilateur.

2.3.1 L'opérateur d'échantillonnage `when`

L'opération `when` est un échantillonneur qui permet à des processus rapides de communiquer avec des processus lents en extrayant des sous-flots de flots en fonction d'une condition, c'est-à-dire d'un flot booléen.

c	f	t	f	t	f	t	...
x	x_0	x_1	x_2	x_3	x_4	x_5	...
x when c		x_1		x_3		x_5	...
x whenot c	x_0		x_2		x_4		...

On voit ici que le flot `x when c` est plus lent que le flot `x`. En supposant que `x` soit produit à une certaine horloge ck, nous dirons que `x when c` est produit à l'horloge ck `on` c.

Ainsi, la fonction `sum` qui calcule la somme de son entrée (i.e., $s_n = \Sigma_{i=0}^{n} x_i$) peut être utilisée à un rythme plus lent en échantillonnant son flot d'entrée:

```
let node sum x = s where rec s = x -> pre s + x
let node sampled_sum x c = sum (x when c)

val sampled_sum :  int -> bool => int
val sampled_sum :: 'a -> (_c0:'a) -> 'a on _c0
```

Alors que la signature de type définit la valeur d'un flot, la signature d'horloge est une abstraction du comportement temporel des programmes: elle indique les instants où la valeur d'un flot est disponible pour être consommée par un processus. Ainsi, `sampled_sum` a une horloge fonctionnelle qui suit la structure de son type (de données) et qui, en notation logique, s'écrit $\forall\alpha.\forall c_0.\alpha \rightarrow (c_0 : \alpha) \rightarrow \alpha$ `on` c_0. Cette signature indique que pour toute horloge α et flot booléen c_0, si le premier argument de la fonction est d'horloge α, que le deuxième argument égal à c_0 est d'horloge α, alors le résultat a l'horloge α `on` c_0 (les variables des fonctions sont renommées dans la signature pour éviter les conflits de noms). Une expression sur l'horloge α `on` c_0 est présente lorsque l'horloge α est vraie et que le flot booléen c_0 est présent et vrai. Un expression ayant l'horloge α `on` c_0 a donc un rythme plus lent qu'une expression sur l'horloge α. Pour revenir à notre exemple, la sortie de `sum (x when c)` est présente seulement aux instants où `c` est vrai.

Cette somme échantillonnée peut maintenant être instanciée avec une horloge particulière. Par exemple:

```
let clock ten = count 10 true
let node sum_ten x = sampled_sum x ten

val ten :  bool
val ten :: 'a
val sum_ten :  int => int
val sum_ten :: 'a -> 'a on ten
```

Le mot-clef `clock` permet d'introduire un nom d'horloge (ici, `ten`) à partir d'un flot booléen. Ce nom d'horloge peut alors être utilisé pour échantillonner un flot.

Les horloges permettent d'exprimer des structures de contrôle sur des blocs flot de données. Filtrer les entrées d'un noeud sur une condition booléenne, par exemple, revient à exécuter ce noeud seulement lorsque cette condition est vraie. Il est donc important de comprendre leur interaction avec les opérations à mémoire tels que les délais.

`c`	f	t	f	t	f	f	t	...
`1`	1	1	1	1	1	1	1	...
`sum 1`	1	2	3	4	5	6	7	...
`(sum 1) when c`		2		4			6	...
`1 when c`		1		1			1	...
`sum (1 when c)`		1		2			3	...

Ainsi, échantillonner l'entrée d'une fonction séquentielle f ne revient pas, en général, à échantillonner sa sortie, c'est-à-dire, $f(x$ `when` $c) \neq (fx)$ `when` c.

Les horloges peuvent être imbriquées arbitrairement. La description d'une montre, par exemple, s'écrira:

```
let clock sixty = count 60 true
let node hour_minute_second second =
  let minute = second when sixty in
  let hour = minute when sixty in
  hour,minute,second

val hour_minute_second :: 'a -> 'a on sixty on sixty * 'a on sixty * 'a
```

Un flot d'horloge `'a on sixty on sixty` est présent seulement tous les 3 600 instants ce qui correspond à ce que l'on attend.

2.3.2 L'opération de combinaison `merge`

Réciproquement, `merge` permet à des processus rapides de communiquer avec des processus lents en mélangeant deux flots. Les deux arguments d'un `merge` doivent avoir des horloges complémentaires avant d'être combinés.

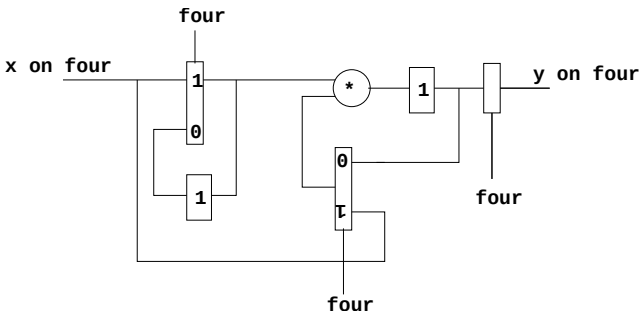


Figure 4: Un exemple de sur-échantillonnage

c	f	t	f	f	f	t	...
x		x_0				x_1	...
y	y_0		y_1	y_2	y_3		...
merge c x y	y_0	x_0	y_1	y_2	y_3	x_1	...

L'opération `merge` permet de définir un bloqueur (l'opérateur `current` de Lustre) qui maintient un flot entre deux échantillons successifs. Ici, `ydef` est une valeur par défaut utilisée lorsqu'aucun échantillon n'a encore été reçu:

```
let node hold ydef c x = y
    where rec y = merge c x ((ydef fby y) whenot c)

val hold :  'a -> bool -> 'a => 'a
val hold :: 'a -> (_c0:'a) -> 'a on _c0 -> 'a
```

2.3.3 Sur-échantillonnage

La combinaison de ces deux opérateurs autorise une forme limitée de sur-échantillonnage, c'est-à-dire de fonctions qui ont un rythme interne supérieur au rythme de leurs entrées/sorties. En ce sens, le langage est strictement plus puissant que Lustre et peut se comparer à Signal qui autorise le sur-échantillonnage.

Le sur-échantillonnage apparaît naturellement lors du découpage d'une tache longue devant être réalisée sur plusieurs cycles de calcul (par exemple lorsque son temps de calcul est trop important ou parce que l'architecture ne dispose pas des ressources suffisantes pour un calcul en un pas de temps). Considérons le calcul de la suite $y_n = (x_n)^5$. Cette suite peut être obtenue en écrivant simplement:

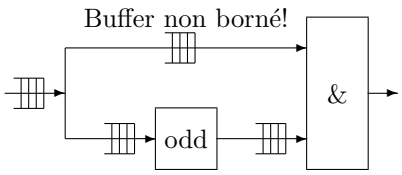
```
let power x = x * x * x * x * x
val power :: 'a -> 'a
```

La sortie est calculée ici au même rythme que celui de son entrée (d'où la signature d'horloge `'a -> 'a`). Quatre multiplications sont nécessaires à chaque cycle. Supposons qu'une seule multiplication ne soit possible à chaque instant (par exemple lorsque l'architecture ne dispose que d'un multiplieur ou que le temps de calcul est trop long). Nous remplaçons ce calcul instantané par une itération dans le temps, en ralentissant l'horloge de `x` d'un facteur quatre[3].

```
let clock four = count 4 true

let node spower x = y where
  rec i = merge four x ((1 fby i) whenot four)
  and o = 1 fby (i * merge four x (o whenot four))
```

[3]Ce découpage d'une tâche longue est inutile dans un exécutif temps-réel préemptif et est confié au système d'exploitation.



`x`	x_0	x_1	x_2	x_3	x_4	x_5	x_6	...
`half`	t	f	t	f	t	f	t	...
`x when half`	x_0		x_2		x_4		x_6	...
`x & (odd x)`	$x_0 \,\&\, x_0$		$x_1 \,\&\, x_2$		$x_2 \,\&\, x_4$		$x_3 \,\&\, x_6$	...

Figure 5: Un programme non synchrone

```
  and y = o when four

val spower :  int => int
val spower :: 'a on four -> 'a on four
```

Le réseau flot de données correspondant est donné dans la figure 4.

`four`	t	f	f	f	t	f	f	f	t	f	f	f	t	f	...
`x`	x_0				x_1				x_2				x_3		...
`i`	x_0	x_0	x_0	x_0	x_1	x_1	x_1	x_1	x_2	x_2	x_2	x_2	x_3	x_3	...
`o`	1	x_0^2	x_0^3	x_0^4	x_0^5	x_1^2	x_1^3	x_1^4	x_1^5	x_2^2	x_2^3	x_2^4	x_2^5	x_3^2	...
`spower` x	1				x_0^5				x_1^5				x_2^5		...
`power` x	x_0^5				x_1^5				x_2^5				x_3^5		...

La fonction `power` ayant la signature d'horloge `'a -> 'a`, la séquence `power x` a donc la même horloge que l'entrée `x`. Ainsi, `spower x` produit la même séquence que `1 fby (power x)`. `spower x` correspond à un raffinement temporel de `1 fby (power x)`. Elle produit la même séquence de valeurs mais à un rythme quatre fois plus lent. Implantée sous forme d'un processus de Kahn connecté à un *buffer* d'entrée, la fonction `spower` correspond à lire une entrée puis à itérer un calcul et à produire un résultat.

Une conséquence importante de la discipline d'horloges et de l'inférence est la possibilité de remplacer toute utilisation de `(1 fby power x)` par `spower x` sans avoir à modifier le reste du programme. Le système global sera alors automatiquement ralenti pour s'adapter aux nouvelles contraintes d'horloges. C'est une propriété importante pour la conception modulaire et la réutilisation de code. Cependant, les horloges de Lucid Synchrone (comme celles de Lustre) pouvant être construites à partir de n'importe quelle expression, le compilateur est incapable d'en tirer des informations quantitatives (qui permettraient ici de démontrer l'équivalence entre `(1 fby power x)` et `spower x`). Autrement dit, `four` est un symbole d'horloge et son caractère périodique est masqué. Des travaux récents permettent d'envisager une extension des langages synchrones avec des horloges périodiques [14, 18].

2.3.4 Contraintes d'horloges et synchronisme

Comme nous l'avons dit, certaines contraintes portant sur les horloges doivent être vérifiées statiquement: les arguments d'un `merge`, par exemple, doivent avoir des horloges complémentaires pour pouvoir être composés, et les arguments des opérations point à point doivent avoir la même horloge.

Quel sens pourrions-nous donner à la composition de programmes ne respectant pas ces contraintes? Illustrons ce point sur l'exemple suivant:

```
let clock half = count 2 true
let node odd x = x when half
let node wrong x = x & (odd x)
```

Ce programme ajoute le flot `x` au sous-flot `x` obtenu en filtrant une entrée sur deux[4]. Ainsi, cette fonction devrait calculer la séquence $(x_n \& x_{2n})_{n \in \mathbb{N}}$. Graphiquement, ce calcul correspond au réseau de Kahn [38] décrit dans la figure 5. Dans ce réseau, l'entrée `x` est dupliquée, l'une des valeur étant connectée à l'opérateur `odd` dont le rôle est d'éliminer une entrée sur deux. Les deux flots sont ensuite reliés à une porte `&` qui consomme un élément de ses entrées à chaque instant. Ce réseau ne peut clairement pas être exécuté sans utiliser de mécanismes de synchronisation par *buffers* et ne pourra donc pas être compilé efficacement. C'est pour cette raison qu'il sera rejeté statiquement. De plus, la taille du *buffer* augmentera au cours du temps et finira par déborder. Cela explique pourquoi ce type d'exemple doit être rejeté dans le cadre de la programmation de systèmes temps réel. On retrouve également une propriété de la théorie des automates: la composition finie d'opérateurs à mémoire bornée (ici un échantillonneur moitié et un opérateur logique) ne préservant pas les longueurs peut ne pas conduire à un calcul à mémoire bornée. L'objectif du calcul d'horloge dans les langages synchrones est donc de rejeter ce type de programmes et de garantir que le caractère temps-réel est conservé par composition [13].

En Lucid Synchrone, les horloges sont des types et le calcul d'horloge prend la forme d'un système d'inférence de types [16, 22]. Lorsque le programme précédent est donné au compilateur, celui-ci produit le message suivant:

```
> let node wrong x = x & (odd x)
>                        ^^^^^^^
This expression has clock 'b on half, but is used with clock 'b.
```

`x` est d'horloge `'b` alors que `odd x` est d'horloge `'b on half`.

La définition formelle des règles d'horloges permet d'établir un résultat de correction: tout programme ayant passé le calcul d'horloge peut être exécuté de manière synchrone. Ce résultat est donc à rapprocher du résultat de correction des types dans les langages ML [50]. Les horloges étant des types, elles servent à décrire de manière abstraite le comportement temporel des composants d'un système. Elles conduisent à une véritable discipline de programmation.

2.4 Valeurs statiques

Les valeurs statiques sont des valeurs constantes au cours du temps et qui servent à définir des systèmes paramétrés, ces paramètres étant fixés une fois pour toute en début d'exécution. Ces valeurs sont introduites au moyen du mot clef `static`:

```
let static m = 100.0
let static g = 9.81
let static mg = m *. g
```

Le compilateur vérifie que les variables déclarées avec le mot clef `static` sont effectivement instanciées avec des valeurs qui le sont.

Le langage tel que nous l'avons vu jusqu'à présent n'est pas très éloigné de Lustre. Les différences essentielles sont pour le moment les inférences de types et d'horloges, qui n'ont donc pas besoin d'être déclarés, ainsi que l'opération `merge`, qui est à notre avis une opération essentielle si l'on veut programmer des systèmes multi-horloges. Nous allons maintenant introduire diverses extensions qui sont spécifiques au langage.

2.5 Types de données et filtrage

Pour l'instant, nous n'avons utilisé que des types de base et des tuples. Le langage permet également de définir des types structurés. On peut définir par exemple un type `number` dont la valeur est soit un entier, soit un flottant. Le type `circle` définit un cercle par les coordonnées de son centre et son rayon.

[4] Dans le vocabulaire du contrôle, `x when half` est un échantillonneur moitié.

```
type number = Int of int | Float of float
type circle = { center: float * float; radius: float }
```

Le langage permet de définir une fonction par cas à l'aide d'un mécanisme de filtrage comme c'est le cas dans les langages ML. Nous l'illustrons sur l'exemple d'un disque tournant et la détection de son sens de rotation. Ce disque est composé de trois sections de couleurs bleue (`Bleu`), rouge (`Rouge`) et verte (`Vert`). Un capteur observe les couleurs successives et doit déterminer si le disque est immobile et sinon, son sens de rotation.

Le sens est dit direct lors d'une succession `Rouge`, `Vert`, `Bleu`..., la direction opposée étant dite indirecte (`Indirect`). À certains instants, la direction peut être indéterminée (`Indetermine`) et le disque peut être immobile (`Immobile`).

```
type color = Bleu | Rouge | Vert
type dir = Direct | Indirect | Indetermine | Immobile

let node direction i = d where
  rec pi = i fby i
  and ppi = i fby pi
  and match ppi, pi, i with
        (Rouge, Rouge, Rouge) | (Bleu, Bleu, Bleu)
      | (Vert, Vert, Vert) ->
            do d = Immobile done
      | (_, Bleu, Rouge) | (_, Rouge, Vert) | (_, Vert, Bleu) ->
            do d = Direct done
      | (_, Rouge, Bleu) | (_, Bleu, Vert) | (_, Vert, Rouge) ->
            do d = Indirect done
      | _ -> do d = Indetermine done
      end
```

Le comportement est défini par un ensemble de cas en comparant les trois valeurs successives de l'entrée `i`. Remarquons que la définition donnée ici privilégie une détection rapide du sens de rotation et lente de l'arrêt. Chaque cas est défini par un ensemble d'équations définissant des *variables partagées* (ici, la variable `d`). À chaque instant, la construction `match/with` sélectionne le premier motif (de haut en bas) qui filtre la valeur du triplet `(pii, pi, i)` et exécute la branche correspondante. Une seule branche est exécutée à chaque réaction.

On retrouve ici tout l'intérêt de la construction de filtrage: le compilateur est capable de vérifier qu'aucun cas n'a été oublié ou n'est redondant. Dans le compilateur de Lucid Synchrone, cette analyse reprend fidèlement celle utilisée dans le compilateur OCaml [43].

2.6 Une construction pour partager la mémoire

La construction de filtrage est une structure de contrôle dont le comportement est très différent de celui de la construction `if/then/else`. Durant une réaction, une branche seulement est active, les autres restant silencieuses. Les arguments d'une conditionnelle, au contraire, avancent tous au même rythme. La construction de filtrage correspond donc à un `merge` qui combine des flots d'horloges complémentaires.

Se pose alors tout naturellement le problème de l'accès dans une branche à la dernière valeur d'une variable calculée dans une autre branche. Il s'agit d'un problème classique de modélisation de systèmes ayant plusieurs modes de fonctionnement dans un outil graphique flot de données (par exemple, SCADE ou Simulink). Les différents modes sont décrits par des planches flot de données activées de manière exclusives et il faut sortir les mémoires partagées par les différents modes pour qu'elles soient mises à jour avec la dernière valeur calculée. Ceci conduit à une programmation peu élégante, d'où l'idée d'introduire une nouvelle construction permettant d'initialiser et d'accéder à une mémoire partagée[5]. La dernière valeur calculée d'un flot partagé `o` peut être accédée en

[5]L'outil Simulink propose un mécanisme permettant de simplifier la communication entre différents schémas-blocs. Ce mécanisme est basé sur l'utilisation de variables impératives pouvant être lues et modifiées par les différents schémas.

écrivant `last o`. Ceci peut être illustré sur un système très simple formé de deux modes. Le mode *up* incrémente une variable partagée `o` alors que le mode *down* la décrémente.

```
let node up_down m step = o where
  rec match m with
        true -> do o = last o + step done
      | false -> do o = last o - step done
      end
  and last o = 0
```

L'équation `last o = 0` définit une mémoire partagée `last o` initialisée à `0`. La communication entre les deux modes est réalisée en faisant référence à `last o` qui contient la dernière valeur calculée de `o`.

step	1	1	1	1	1	1	1	1	1	1	1	1	1	1	...
m	t	t	t	f	f	t	t	t	f	f	t	t	t	t	...
last o	0	1	2	3	2	1	2	3	4	3	2	3	4	5	...
o	1	2	3	2	1	2	3	4	3	2	3	4	5	6	...

Ce programme a le même comportement que le programme suivant[6]. Ici, nous avons dû placer le calcul de la dernière valeur de `o` à l'extérieur de la structure de contrôle.

```
let node up_down m step = o where
  rec match m with
        true -> do o = last_o + step done
      | false -> do o = last_o - step done
      end
  and last_o = 0 -> pre o
```

Bien que `last o` soit une autre manière de faire référence à la valeur précédente d'un flot et ressemble donc à `pre o`, il y a une différence fondamentale entre les deux. Cette différence est une question d'instant d'observation. Dans les schémas flot de données, `pre` (e) désigne une mémoire locale, c'est-à-dire la dernière valeur de son argument, la dernière fois qu'il a été observé. Si `pre` (e) apparaît dans une structure de bloc qui n'est exécutée que de temps en temps, disons à une horloge c, cela signifie que l'argument e est calculé seulement lorsque l'horloge c est vraie. Au contraire, `last` o désigne la dernière valeur de la variable o à l'instant où la variable est définie. `last` o s'applique donc seulement à une variable et non à une expression. `last` o permet de communiquer une valeur entre deux modes et c'est la raison pour laquelle elle désigne une mémoire partagée. La sémantique du programme est donc précisément définie par l'écriture suivante.

```
let node up_down m step = o where
  rec o = merge c ((last_o when c) + (step when c))
                  ((last_o whenot c) - (step whenot c))
  and clock c = m
  and last_o = 0 -> pre o
```

Pour terminer, on remarquera que l'introduction de mémoires partagées permet de compléter implicitement les flots avec leur valeur précédente. Ainsi, l'absence d'une équation définissant une variable `x` correspond à l'ajout implicite d'une équation `x = last x`. Cette caractéristique s'avère très utile en pratique lorsque chacun des modes définit plusieurs flots.

2.7 Signaux

Lorsque l'on compare Lustre et Esterel, on note que le premier manipule des flots alors que le second manipule des signaux. Esterel distingue deux types de signaux: les signaux purs et les signaux valués. Un signal pur est essentiellement un flot booléen, vrai lorsque le signal est émis et faux sinon. Un signal valué est un objet plus complexe que le flot: il peut être émis ou pas, et

[6] C'est bien plus que cela puisque le compilateur traduit le premier dans le second.

lorsqu'il est émis, il transporte une valeur. Un signal peut présenter un avantage important pour le programmeur: il n'existe que lorsqu'il est explicitement émis alors que la valeur d'un flot doit être définie dans tous les cas.

Nous introduisons donc maintenant un mécanisme permettant de manipuler des signaux valués dans un cadre flot de données. Ces signaux sont construits et accédés grâce à deux constructions, `emit` et `present`. Un signal valué est un couple composé (1) d'un flot booléen c indiquant les instants où le signal est présent et (2) un flot échantillonné sur cette condition c. La signature d'horloge d'un signal x est une paire dépendante $\Sigma(c : \alpha).\alpha$ `on` c formée d'un flot booléen c d'horloge α et d'un second flot contenant la valeur et d'horloge α `on` c[7]. Nous notons cette signature α `sig`.

2.7.1 Les signaux vus comme des abstractions d'horloges

Un signal peut être construit à partir d'un flot échantillonné par abstraction de son horloge interne.

```
let node within min max x = o where
  rec clock c = (min <= x) & (x <= max)
  and emit o = true when c

val within :  'a -> 'a -> 'a => bool sig
val within :: 'a -> 'a -> 'a -> 'a sig
```

Cette fonction calcule une condition `c` et un flot échantillonné `true when c`. L'équation `emit o = true when c` définit un signal `o` présent et égal à la valeur vraie lorsque `c` est vrai. La construction `emit` encapsule la valeur émise avec son information d'horloge. Ce mécanisme correspond à une forme limitée de type existentiel [54].

2.7.2 Tester la présence et filtrer des signaux

Il est possible de tester la présence d'une expression de signal e en écrivant l'expression booléenne `?`e. Le programme suivant, par exemple, compte le nombre d'instants où `x` est émis.

```
let node count x = cpt where
  rec cpt = if ?x then 1 -> pre cpt + 1 else 0 -> pre cpt

val count :  'a sig => int
val count :: 'a sig -> 'a
```

Le langage fournit un mécanisme plus général permettant de tester la présence de plusieurs signaux et d'accéder à leurs valeurs. Ce mécanisme est proche de la construction de filtrage de ML et reprend la notation introduite dans le Join-Calcul [27].

Le programme suivant, par exemple attend deux signaux `x` et `y` et retourne un entier égal à la somme de `x` et `y` lorsque les deux signaux sont émis. Il retourne la valeur de `x` lorsque `x` est émis seulement, la valeur de `y` lorsque `y` est émis seulement et `0` sinon.

```
let node sum x y = o where
  present
    x(v) & y(w) -> do o = v + w done
  | x(v1) -> do o = v1 done
  | y(v2) -> do o = v2 done
  | _ -> do o = 0 done
  end

val sum :  int sig -> int sig => int
val sum :: 'a sig -> 'a sig -> 'a
```

[7] Dans le vocabulaire des circuits, un signal correspond à une paire formée d'une valeur booléenne et d'un fil *enable* de contrôle indiquant les instant où la valeur est pertinente [58].

La construction `present` est composée de plusieurs cas formés d'un filtre sur des signaux et d'un traitement. Chaque filtre est traité séquentiellement. Ainsi, le filtre `x(v) & y(w)` est vérifié lorsque les deux signaux `x` et `y` sont présents et leurs valeurs sont alors liées aux identificateurs `v` et `w` qui peuvent être utilisées dans la branche correspondante. Si ce filtre n'est pas vérifié, le second est testé, etc. Le dernier cas `_` définit le traitement par défaut. Dans le filtre `x(v) & y(w)`, `x` et `y` sont des expressions s'évaluant en des signaux alors que `v` et `w` sont des motifs. Ainsi, le filtre `x(42) & y(w)` se lit: attendre que `x` soit présent et s'évalue à `42` et que `y` soit présent.

En utilisant les signaux, il est possible de mimer la construction `default` du langage Signal. `default x y` émet la valeur de `x` lorsque `x` est présent et la valeur de `y` lorsque `x` est absent et `y` est présent. `o` étant un signal, il ne conserve implicitement pas sa valeur dans le dernier cas (`x` et `y` absents) et est supposé absent.

```
let node default x y = o where
  present
    x(v) -> do emit o = v done
  | y(v) -> do emit o = v done
  end

val default :  'a -> 'a => 'a
val default :: 'a sig -> 'a sig -> 'a sig
```

Il s'agit seulement d'une simulation puisque les informations d'horloge — les instants précis où `x`, `y` et `default x y` sont émis — sont maintenant masquées. Le compilateur n'est plus capable d'établir que `o` est émis seulement lorsque `x` ou `y` sont présents comme le fait le calcul d'horloge de Signal. L'utilisation des signaux permet cependant de retrouver, dans un cadre flot de données, un style de programmation propre à Esterel, qui se révèle confortable pour programmer des systèmes à contrôle prépondérant.

La construction de filtrage de signaux est sûre dans la mesure où il est possible d'accéder à la valeur d'un signal seulement aux instants où il est émis. C'est une différence importante avec Esterel où la valeur d'un signal est implicitement maintenue et peut être accédée même lorsqu'elle n'est pas émise.

2.8 Machines à états et systèmes mixtes

Le langage permet de définir des machines à états (ou automates) afin de décrire directement des systèmes à contrôle prépondérant (*control dominated systems*). Ces machines à états peuvent être composées en parallèle avec des équations ou d'autres machines à états et peuvent être imbriquées arbitrairement.

Un automate est un ensemble d'états et de transitions. Un état est formé d'un ensemble d'équations exécutées lorsque l'état est actif, dans l'esprit des automates de modes de Maraninchi et Rémond [45]. Deux types de transitions, dites fortes ou faibles, peuvent être déclenchées à partir d'un état et, pour chacune d'entre elles, l'état destination peut être entré par réinitialisation ou par histoire.

2.8.1 Préemption faible et préemption forte

Dans un monde synchrone, on peut envisager au moins deux types de transitions à partir d'un état: une transition est dite faible lorsque celle-ci est effectuée à la fin de la réaction ou forte, lorsqu'elle est effectuée immédiatement, en début de réaction.

Voici un exemple d'automate à deux états contenant une transition faible:

```
let node weak_switch on_off = o where
  automaton
    Off -> do o = false until on_off then On
  | On -> do o = true until on_off then Off
  end
```

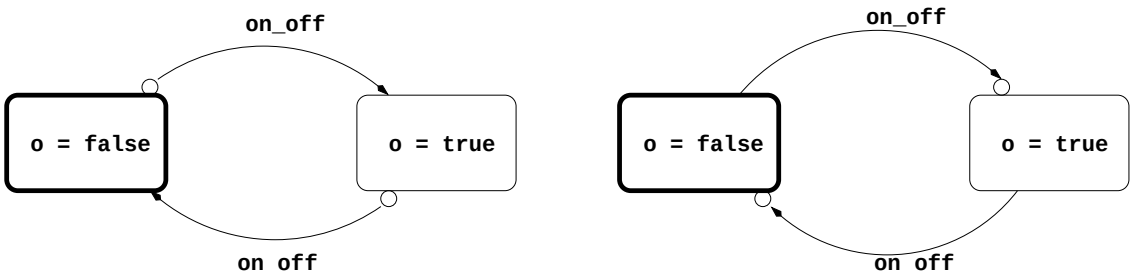


Figure 6: Automates avec transitions fortes ou faibles

Un automate est composé d'un ensemble d'états (dont le premier dans l'ordre désigne l'état initial), chaque état définissant un ensemble de variables partagées. L'automate ci-dessus est composé de deux états `Off` et `On` définissant chacun la valeur courante de `o`. Le mot-clé `until` indique que `o` est défini par l'équation `o = false` de l'état initial `Off` jusqu'à l'instant (inclus) où `on_off` est vrai. Après cet instant, l'état actif sera l'état `On`. Un automate avec transition faible correspond donc au modèle classique des automates de Moore. Au contraire, la fonction suivante produit la valeur vraie dès que l'entrée `on_off` est vraie. Ici, la valeur de `o` est définie par l'équation `o = false` à moins que la condition `on_off` ne soit vérifiée. La condition est donc testée avant d'exécuter les définitions de l'état courant: elle est dite forte.

```
let node strong_switch on_off = o where
  automaton
    Off -> do o = false unless on_off then On
  | On -> do o = true unless on_off then Off
  end
```

La représentation graphique de ces deux automates est présentée dans la figure 6. Nous reprenons ici la notation introduite par Jean-Louis Colaço et issue des SyncCharts [1]: une transition forte est représentée par une flèche commençant par un rond, indiquant ainsi que la condition est évaluée au même instant que le corps de l'état cible. Réciproquement, une transition faible est représentée par une flèche terminée par un rond, indiquant ainsi que la condition est évaluée au même instant que le corps de l'état source.

`on_off`	f	f	t	f	f	t	f	t	...
`weak_switch on_off`	f	f	f	t	t	t	f	f	...
`strong_switch on_off`	f	f	t	t	t	f	f	t	...

On peut ainsi remarquer que pour tout flot booléen `on_off`, la séquence de valeur produite par l'expression `weak_switch on_off` est égale à celle de l'expression `strong_switch (false -> pre on_off)`.

2.8.2 ABRO et la réinitialisation modulaire

L'ajout des automates à un langage flot de données permet de retrouver des traits propres au langage Esterel. Nous l'illustrons sur le (fameux) exemple ABRO montrant l'intérêt de la composition synchrone et de la préemption [7]. Sa spécification est la suivante:

> Attendre la présence des événements `a` et `b` et émettre `o` a l'instant précis où les deux événements ont été reçus. Réinitialiser le comportement à chaque occurrence de l'événement `r`.

On définit d'abord un noeud `expect` permettant d'attendre la présence d'un événement et de maintenir la valeur vraie à partir de cet instant.

```
let node expect a = o where
  automaton
```

```
    S1 -> do o = false unless a then S2
  | S2 -> do o = true done
  end

let node abo a b = (expect a) & (expect b)
```

Le noeud `abo` rend la valeur vraie et la maintient dès que les entrées `a` et `b` sont vraies. C'est donc la mise en parallèle de deux automates scrutant les flots `a` et `b` composée avec un et logique. Pour obtenir le noeud `abro`, nous construisons un nouvel automate à un seul état effectuant une transition à chaque fois que la condition `r` est vraie. L'état cible est alors réinitialisé: tout flot ou automate repart dans sa configuration initiale. Cette réinitialisation est indiquée par le mot-clef `then`.

```
let node abro a b r = o where
  automaton
    S -> do o = abo a b unless r then S
  end
```

Le langage fournit une construction `reset/every` comme raccourci d'un tel automate. On écrira:

```
let node abro a b r = o where
  reset
    o = abo a b
  every r
```

2.8.3 Définitions locales à un état

Chaque état d'un automate est composé d'un ensemble d'équations définissant des variables partagées. Il est possible de définir des variables locales à un état. L'utilisation de ces variables est alors limitée au calcul des variables partagées de l'état courant et des transitions faibles. Elles ne peuvent pas être utilisées dans les transitions fortes puisque celles-ci sont testées en début de réaction. Le compilateur détecte statiquement de telles situations et rejette le programme.

Le programme suivant maintient la valeur vraie pendant une durée `d1`, la valeur fausse pendant une durée `d2` puis recommence.

```
let node alternate d1 d2 = status where
  automaton
    True ->
      let rec c = 1 -> pre c + 1 in
      do status = true
      until (c = d1) then False
  | False ->
      let rec c = 1 -> pre c + 1 in
      do status = false
      until (c = d2) then True
  end
```

L'état `True` définit une variable `c`, locale à l'état et pouvant être utilisée pour le calcul de la transition faible `c = max`. Ceci n'introduit pas de problème de causalité puisque la condition n'est exécutée qu'en fin de réaction et définit l'état du système à l'instant suivant.

2.8.4 Communication entre états et mémoires partagées

Dans les exemples que nous avons vu, il n'y a pas de communication entre les valeurs calculées dans chaque état. La nécessité de communiquer des valeurs entre états apparaît naturellement dans un système ayant plusieurs modes de fonctionnement. Cette communication sera réalisée en utilisant la construction `last` et nous l'illustrons sur un système formé de trois états.

```
let node up_down go = o where
  rec automaton
        Init ->
           do o = 0 until go then Up
      | Up ->
          do o = last o + 1
          until (o >= 5) then Down
      | Down ->
          do o = last o - 1
          until (o <= - 5) then Up
      end
```

Ce programme calcule donc la suite:

go	*f*	*f*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*	*t*
o	0	0	0	1	2	3	4	5	4	3	2	1	0	-1	-2	-3	-4	-5	-4	-3	-2

La transition de sortie de l'état initial `Init` étant une transition faible, la valeur de `last o` sera nécessairement définie lors de l'entrée dans l'état `Up`. `last o` contient à chaque instant la dernière valeur calculée de la variable partagée `o`.

2.8.5 Actions par défaut et le rôle particulier de l'état initial

Chaque état d'un automate est composé d'un ensemble d'équations de flots portant sur des variables partagées. Jusque-là, nous avons considéré des automates où les variables partagées étaient définies dans chacun des états. En pratique, il est souvent pénible de devoir définir la valeur d'une variable partagée dans chacun des états, d'où la nécessité d'un mécanisme permettant de leur donner une valeur par défaut. Le choix fait dans Lucid Synchrone consiste à maintenir la dernière valeur calculée. Considérons l'exemple d'un bouton permettant d'ajuster une valeur entière[8]:

```
let node adjust p m = o where
  rec last o = 0
  and automaton
        Idle ->
           do unless p then Incr unless m then Decr
      | Incr ->
           do o = last o + 1 unless (not p) then Idle
      | Decr ->
           do o = last o - 1 unless (not m) then Idle
      end
```

L'absence d'équation dans l'état `Idle` signifie que `o` garde implicitement sa valeur. Autrement dit, l'équation `o = last o` est implicitement ajouté à cet état.

2.8.6 Réinitialiser ou prolonger un état

Lors d'une transition, il est possible de spécifier si l'état vers lequel s'effectue la transition est réinitialisé ou pas. Les transitions considérées jusque-là étaient des transitions réinitialisant l'état d'arrivée. Le langage permet également de continuer l'exécution d'un état dans la configuration où il se trouvait à son dernier instant d'activation. On écrira, par exemple:

```
let node up_down () = o where
  rec automaton
        Up -> do o = 0 -> last o + 1 until (o >= 5) continue Down
      | Down -> do o = last o - 1 until (o <= 5) continue Up
      end
```

Le chronogramme est alors:

o	0	1	2	3	4	5	4	3	2	1	0	-1	-2	-3	-4	-5	-4	...

[8]Cet exemple est dû à Jean-Louis Colaço et Bruno Pagano.

Notons que `last o` est nécessairement défini en entrant dans l'état `Down` puisque la transition de l'état `Up` est une transition faible. Son remplacement par une transition forte (`unless`) lève une erreur d'initialisation.

2.9 Machines d'états paramétrées

Dans les exemples considérés jusque là, un automate est composé d'un ensemble d'états et de transitions. Il est possible d'envisager des automates un peu plus expressifs où les états sont paramétrés par des valeurs pouvant être transmises au moment de la transition. Ce mécanisme permet de réduire le nombre d'états de l'automate et d'exprimer dans un cadre uniforme un traitement particulier à l'entrée dans l'état (par exemple les *transition on entry* de Stateflow).

Le programme suivant compte le nombre d'occurrences de `x`.

```
let node count x = o where
  automaton
    Zero -> do o = 0 until x then Plus(1)
  | Plus(v) -> do o = v until x then Plus(v+1)
  end
```

Nous considérons maintenant la programmation d'un contrôleur de souris dont la spécification est la suivante:

> Produire l'événement `double` lorsque deux événements `click` ont été reçus en moins de quatre `top`. Émettre `simple` si un seul événement `click` a été reçu

Le contrôleur de souris peut être décrit par un automate à trois états:

```
let node counting e = cpt where
  rec cpt = if e then 1 -> pre cpt + 1 else 0 -> pre cpt

let node controler click top = (simple, double) where
  automaton
    Await ->
     do simple = false and double = false
     until click then One
  | One ->
     do simple = false and double = false
     unless click then Emit(false, true)
     unless (counting top = 4) then Emit(true, false)
  | Emit(x1, x2) ->
     do simple = x1 and double = x2
     until true then Await
  end
```

Ce contrôleur attend la première occurrence de `click` puis entre dans l'état `One` et commence à compter le nombre de `top`. Cet état peut être préempté fortement lorsqu'un second `click` est reçu ou que la condition `counting top = 4` est vraie. Par exemple, si `click` est vrai, le contrôle passe immédiatement dans l'état `Emit(false, true)`, donnant les valeurs initiales `false` et `true` aux variables `x1` et `x2`. Ainsi, à cet instant précis, `simple = false` et `double = true`. Puis le contrôleur retourne dans l'état initial `Await` à l'instant suivant.

Cet exemple illustre une caractéristique essentielle des machines d'états introduites dans Lucid Synchrone: un seul ensemble d'équations est exécuté au cours d'une réaction. Cependant, il est possible d'effectuer (au plus) une préemption forte suivie d'une préemption faible au cours d'une réaction, et c'est exactement ce que nous avons fait ici. À la différence d'autres formalismes tels que les StateCharts [35] ou les SyncCharts [1], il est impossible de traverser un nombre arbitrairement grand d'états au cours d'une réaction. Ceci conduit à une programmation et une mise au point souvent plus simples.

2.10 Machines d'états et signaux

Dans les automates considérés jusque-là, les conditions d'échappement sont des conditions booléennes. Le langage fournit un mécanisme plus général permettant de tester la présence (et d'accéder à la valeur) d'un signal lors d'une transition.

Nous l'illustrons sur l'exemple d'un système à deux signaux d'entrées valués `low` et `high` et produisant un flot `o`.

```
let node switch low high = o where
  rec automaton
        Init -> do o = 0 then Up(1)
      | Up(u) ->
          do o = last o + u
          unless low(v) then Down(v)
      | Down(v) ->
          do o = last o - v
          unless high(w) then Up(w)
      end

val switch :  'a sig -> 'a sig => 'a
val switch :: 'a sig -> 'a sig -> 'a
```

La condition `unless low(v) then Down(v)` se lit: aller dans l'état paramétré `Down(v)` lorsque le signal `low` est présent et de valeur `v`. La construction `do o = 0 then Up(1)` est un raccourci pour `do o = 0 until true then Up(1)`.

high	*abs*	3	*abs*	*abs*	*abs*	*abs*	*abs*	*abs*	*abs*	2	*abs*	*abs*	*abs*	*abs*	*abs*	*abs*
low	*abs*	*abs*	*abs*	1	*abs*	9	*abs*	*abs*	2	4	*abs*	*abs*	*abs*	1	*abs*	5
o	0	1	2	1	0	-1	-2	-3	-4	-2	0	2	4	3	2	1

La combinaison des automates et des signaux permet de revoir la spécification du contrôleur de souris et d'écrire:

```
type e = Simple | Double

let node counting e = o where
  rec o = if ?e then 1 -> pre o + 1 else 0 -> pre o

let node controler click top = e where
  automaton
    Await ->
     do until click(_) then One
  | One ->
     do unless click(_) then Emit Double
     unless (counting top = 4) then Emit Simple
  | Emit(x) ->
     do emit e = x then Await
  end

val controler :  'a sig -> 'b sig => 'c sig
val controler :: 'a sig -> 'a sig -> 'a sig
```

Remarquons qu'aucune valeur n'est calculée dans les états `Await` et `One`. En écrivant `emit o = x`, le programmeur indique que `o` est un signal, ce qui n'impose pas de le définir dans chacun des état (ou de compléter implicitement sa valeur courante avec la dernière valeur calculée `last o`). Ici, le signal `o` est seulement émis dans l'état `Emit`. Sinon, il est supposé absent.

L'utilisation combinée des signaux et des types somme présente un intérêt par rapport à la représentation des événements par des flots booléens: ici, la sortie `o` a seulement trois valeurs possibles (correspondant à `Simple`, `Double` ou absent) alors qu'elle en avait quatre dans la version purement booléenne.

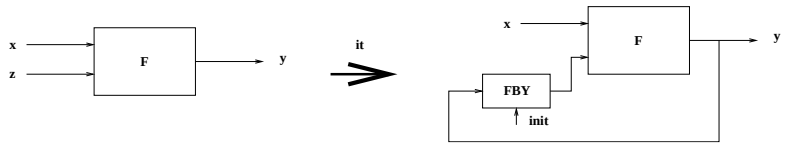


Figure 7: Un itérateur

2.11 Traits d'ordre supérieur

Le langage étant fonctionnel, les fonctions sont des objets de base qui peuvent être donnés en argument à des fonctions ou retournés par des fonctions. Par exemple, la fonction `iter` est un itérateur série qui itère une fonction sur un flot de valeurs. Sa représentation graphique est donnée figure 7.

```
let node iter init f x = y where
  rec y = f x (init fby y)

val iter : 'a -> ('b -> 'a -> 'a) => 'a
val iter :: 'a -> ('b -> 'a -> 'a) -> 'a
```

de sorte que:

```
let node sum x = iter 0 (+) x
let node mult x = iter 1 ( * ) x
```

La signature de type de `iter` indique que la fonction `f` est une fonction combinatoire. Il est possible de définir un opérateur de rebouclage plus général paramétré par une fonction séquentielle en écrivant plutôt:

```
let node iter init f x = y where
  rec y = run (f x) (init fby y)

val iter : 'a -> ('b -> 'a => 'a) 'b => 'a
val iter :: 'a -> ('b -> 'a -> 'a) -> 'b -> 'a
```

Le mot-clef `run` indique que son premier argument `(f x)` est une fonction à mémoire et a donc un type de la forme $t_1 \Rightarrow t_2$ plutôt que $t_1 \rightarrow t_2$. Nous reviendrons dans la partie 3 sur les problèmes de synthèse de type et justifierons ce choix.

L'ordre supérieur est une manière naturelle de définir de nouvelles primitives à partir de primitives existantes. Par exemple, la condition d'activation est un opérateur primitif dans des outils tels que SCADE/Lustre ou Simulink. Une condition d'activation prend en argument une fonction `f`, une condition `c`, une valeur par défaut `default`, une entrée `input` et calcule `f(input when c)`. Elle replace alors le résultat sur l'horloge de l'entrée `input`.

```
let node condact f c default input = o where
  rec o = merge c (run f (input when c))
                  ((default fby o) whenot c)

val condact :  ('a => 'b) -> bool -> 'b -> 'a -> 'b
val condact ::
    ('a on _c0 -> 'a on _c0) -> (_c0:'a) -> 'a -> 'a -> 'a
```

En utilisant la primitive `condact`, il est possible de définir un opérateur classique disponible à la fois dans la librairie SCADE et dans la librairie *digital* de Simulink. Sa représentation graphique en SCADE est donnée dans la figure 1. Cet opérateur détecte un front montant (passage d'une transition fausse à vraie). La sortie devient vraie dès qu'une transition a été détectée et est maintenue pendant `number_of_cycle` cycles. La sortie est initialement fausse et un front montant arrivant alors que la sortie est vraie est détecté.

$$l\frac{d^2\theta}{dt^2} = (sin(\theta)(\frac{d^2y_0}{dt^2} + g)) - (cos(\theta)\frac{d^2x_0}{dt^2})$$
$$x = x_0 + lsin(\theta)$$
$$y = y_0 + lcos(\theta)$$

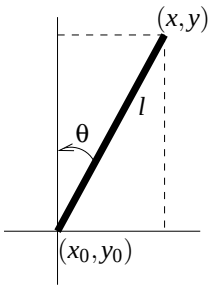


Figure 8: Le pendule inversé

```
let node count_down (res, n) = cpt where
  rec cpt = if res then n else (n -> pre (cpt - 1))

let node rising_edge_retrigger rer_input number_of_cycle = rer_output
  where
    rec rer_output = (0 < v) & (c or count)
    and v = condact count_down clk 0 (count, number_of_cycle)
    and c = false fby rer_output
    and clock clk = c or count
    and count = false -> (rer_input & pre (not rer_input))
```

L'intérêt de l'ordre supérieur est multiple. Pour l'écrivain d'un compilateur, il permet de se concentrer sur un noyau de langage réduit en ramenant de nombreuses constructions aux constructions de ce noyau. Pour le programmeur, il offre la possibilité de définir ses propres librairies de composants génériques. Dans Lucid Synchrone, l'ensemble des analyses de programmes (typage, horloges, etc.) est compatible avec l'ordre supérieur.

2.12 Deux exemples classiques

Nous terminons cette présentation du langage par deux exemples classiques. Le premier est celui du pendule inversé programmé dans un style purement flot de données. Le second est un contrôleur simple de chaudière permettant d'illustrer la combinaison du flot de données et des automates.

2.12.1 Le pendule inversé

Considérons la modélisation d'un pendule inversé. Le pendule a une longueur l, sa base contrôlée manuellement est de coordonnées x_0 et y_0 et il forme un angle θ avec la verticale. La loi du pendule est donnée dans la figure 8.

On commence par réaliser un module synchrone `Misc` permettant d'intégrer et de dériver un flot (la notation `*.` désigne la multiplication flottante).

```
(* module Misc *)
let node integr t x' = let rec x = 0.0 -> t *. x' +. pre x in x
let node deriv t x = 0.0 -> (x -.(pre x))/. t
```

Le module principal est défini ainsi:

```
(* module Pendulum *)
let static dt = 0.05 (* pas d'echantillonnage *)
let static l = 10.0 (* longueur du pendule *)
let static g = 9.81 (* acceleration *)

let node integr x' = Misc.integr dt x'
let node deriv x = Misc.deriv dt x

(* l'equation du pendule *)
```

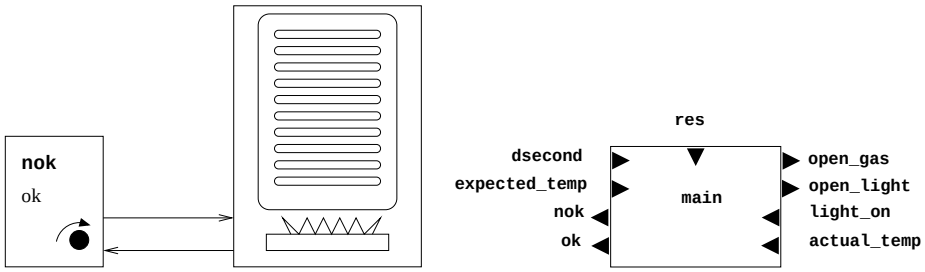


Figure 9: Un contrôleur de chaudière

```
let node equation x0'' y0'' = theta where
  rec theta = integr (integr ((sin thetap) *. (y0'' +. g)
                    -. (cos thetap) *. x0'') /. l)
  and thetap = 0.0 fby theta

let node position x0 y0 =
  let x0'' = deriv (deriv x0)  in
  let y0'' = deriv (deriv y0)  in

  let theta = equation x0'' y0'' in

  let x = x0 +. l *. (sin theta)  in
  let y = y0 +. l *. (cos theta)  in
  Draw.make_pend x0 y0 x y

let node main () =
  let x0,y0 = Draw.mouse_pos () in
  let p = Draw.position x0 y0 in
  Draw.clear_pendulum (p fby p);
  Draw.draw_pendulum p;;
```

La notation pointée `Misc.integr dt x'` fait référence à la fonction `integr` définie dans le module `Misc`. On remarque la grande proximité entre l'écriture de la loi du pendule et la fonction `equation` qui la réalise. Le modèle flot de données synchrone est particulièrement bien adapté à la programmation de tels systèmes.

Ce programme illustre également la communication entre Lucid Synchrone et le langage hôte (ici OCaml) dans lequel certaines fonctions auxiliaires sont écrites. Ici, le module `Draw` exporte un certain nombre de fonctions auxiliaires (une fonction de création d'un pendule `make_pend`, une fonction `mouse_pos` retournant la position de la souris au cours du temps et des fonctions d'affichage).

2.12.2 Un contrôleur de chaudière

Le second exemple est celui d'un controleur de chaudière à gaz représenté dans la figure 9.

Le panneau avant de la chaudière dispose d'un voyant indiquant un fonctionnement normal et un voyant indiquant un arrêt de sécurité. Dans ce cas, la chaudière s'arrête et ne peut être redémarrée qu'en appuyant sur un bouton de redémarrage. De plus, un bouton permet à l'utilisateur d'indiquer la température souhaitée.

Le contrôleur a donc les entrées suivantes: une entrée `res` utilisée pour redémarrer la chaudière; une entrée `expected_temp` indiquant la température souhaitée (la chaudière est supposée maintenir cette température); l'entrée `actual_temp` est la température réelle de l'eau mesurée par un capteur; `light_on` indique qu'une flamme est allumée; `dsecond` est un flot booléen donnant le rythme de base du système. Ce contrôleur produit plusieurs sorties. `open_light` permet d'allumer la flamme;

`open_gas` commande l'ouverture de la vanne de gaz; `ok` indique le status de la chaudière alors que `nok` indique qu'un problème est survenu.

L'objectif de ce contrôleur est de maintenir la température de l'eau proche de la température souhaitée. De plus, lorsque l'eau doit être chauffée, le contrôleur allume le gaz et la flamme pendant au plus cinq dixièmes de secondes. Lorsque la flamme est allumée, seule la vanne de gaz doit être maintenue ouverte. Si la flamme n'a pu être allumée au bout de ce délai, le système s'arrête durant un dixième de secondes puis recommence à nouveau. Si au bout de trois essais l'allumage a échoué, la chaudière est arrêtée en position de sécurité. Seul le bouton `res` peut redémarrer la chaudière.

```
let static low = 4
let static high = 4
let static delay_on = 5 (* en dixièmes de secondes *)
let static delay_off = 1

let node count d t = ok where
  rec ok = cpt = 1
  and cpt = d -> if t & not (pre ok) then pre cpt - 1 else pre cpt

let node edge x = false -> not (pre x) & x
```

Le noeud suivant permet de décider de l'allumage de la chaudière. Nous introduisons pour cela un mécanisme d'*hysteresis* pour éviter les oscillations [41]. `low` et `high` sont deux seuils. La première version de ce noeud est purement flot de données alors que la seconde (équivalente) utilise la construction d'automate.

```
let node heat expected_temp actual_temp = on_heat where
  rec on_heat = if actual_temp <= expected_temp - low then true
                else if actual_temp >= expected_temp + high then false
                else false -> pre on_heat

let node heat expected_temp actual_temp = on_heat where
  rec automaton
        False ->
          do on_heat = false
          unless (actual_temp <= expected_temp - low) then True
      | True ->
          do on_heat = true
          unless (actual_temp >= expected_temp + high) then False
      end
```

Le noeud permettant d'allumer la flamme pendant `delay_on` dixièmes de secondes, de l'éteindre pendant `delay_off` dixièmes de secondes puis de recommencer s'écrit:

```
let node command dsecond = (open_light, open_gas) where
  rec automaton
        Open ->
          do open_light = true
          and open_gas = true
          until (count delay_on dsecond) then Silent
      | Silent ->
          do open_light = false
          and open_gas = false
          until (count delay_off dsecond) then Open
      end
```

Le contrôle de l'ouverture de la flamme et du gaz peuvent maintenant s'écrire ainsi:

```
let node light dsecond on_heat light_on =
                           (open_light, open_gas, nok) where
  rec automaton
        Light_off ->
```

```
        do nok = false
        and open_light = false
        and open_gas = false
        until on_heat then Try
    | Light_on ->
        do nok = false
        and open_light = false
        and open_gas = true
        until (not on_heat) then Light_off
    | Try ->
        do
          (open_light, open_gas) = command dsecond
        until light_on then Light_on
        until (count 3 (edge (not open_light))) then Failure
    | Failure ->
        do nok = true
        and open_light = false
        and open_gas = false
        done
    end
```

La fonction principale, enfin, relie les deux composants précédents.

```
let node main dsecond res expected_temp actual_temp light_on =
                              (open_light, open_gas, ok, nok) where
  rec reset
        on_heat = heat expected_temp actual_temp
      and
        (open_light, open_gas,nok) = light dsecond on_heat light_on
      and
        ok = not nok
      every res
```

Dans tous ces exemples, nous avons seulement décrit le noyau réactif à partir duquel le compilateur produit une fonction de transition. Dans la version actuelle de Lucid Synchrone, cette fonction est écrite en OCaml et peut être liée à d'autres librairies pour obtenir un exécutable final.

3 Discussion

Nous revenons ici sur les travaux reliés à Lucid Synchrone et en particulier les plongements de langages dédiés à la description de circuits ou de systèmes réactifs dans les langages fonctionnels; les techniques de typage utilisées dans le langage et les travaux sur la description de systèmes mixtes.

3.1 Programmation fonctionnelle pour le réactif ou la description de circuits

L'intérêt d'utiliser un langage purement fonctionnel pour décrire des circuits synchrones a été identifié très tôt par Mary Sheeran dans μFP [55]. Depuis, de nombreux langages ou librairies ont été plongés dans le langage généraliste Haskell pour décrire des circuits (Hydra [49], Lava [8]), des architectures de processeurs (par exemple, Hawk [48]), des systèmes réactifs (Fran [26], frp [60]) et des langages fonctionnels dédiés à la conception de circuit ont été proposés (par exemple, Jazz [58], ReFleCT [39]). Les circuits et les systèmes dynamiques peuvent être modélisés directement en Haskell en utilisant des modules définissant les opérations de base (par exemple, délais, opérations logiques, transformateurs de suites), conduisant à une programmation très proche de celle offerte dans les langages synchrones tels que Lustre ou Lucid Synchrone. Le plongement de langages dédiés dans un langage tel que Haskell permet de profiter de la puissance d'expression du langage hôte (typage, structures de données et structures de contrôle, évaluation) et évite d'avoir à

écrire un compilateur dédié. Surtout, le mécanisme de classes de types de Haskell [31] permet de changer aisément de représentation des flots pour obtenir une vérification de propriété, une simulation ou une compilation. Les techniques *multi-stage* [56] permettent, de la même manière, de décrire un langage dédié. Cette approche par plongement dans un langage généraliste est très bien adaptée dans le cas de langages de descriptions de circuits où le résultat de la compilation est essentiellement un réseau d'opérateurs booléens à plat (*net-lists*). Elle est cependant limitée dans le cas d'un langage visant une compilation vers du logiciel (comme c'est principalement le cas de SCADE/Lustre). Même si un réseau d'opérateurs peut ensuite être compilé vers du code séquentiel, le code obtenu est peu efficace à cause de la croissance du code due au dépliage systématique et à la disparition du partage et de la structure de contrôle du programme source. De plus, lorsque les fonctions ne préservant pas les longueurs sont autorisées (comme c'est le cas dans Fran ou frp), le caractère temps réel (exécution en mémoire et temps bornés) n'est pas garanti statiquement[9]. Aucun de ces travaux ne fournit de notion d'horloge, de structure de contrôle mélangeant une programmation par équations et une programmation par automates ni de techniques de compilation ou d'analyses statiques dédiées. Le choix fait dans Lucid Synchrone conduit plutôt à rejeter de nombreux programmes afin de pouvoir obtenir des garanties à la compilation (par exemple, exécution synchrone, absence de boucles de causalité). Les programmes sont ensuite compilés vers du code séquentiel.

3.2 Types, horloges et analyses statiques

Le langage Lucid Synchrone dispose de plusieurs analyses statiques toutes définies sous forme de systèmes de types. Le typage peut être réalisé par un système de types à la ML et être implanté en utilisant les techniques classiques [50]. Dans les premières versions du compilateur, le typage ne faisait pas de distinction entre les fonctions combinatoires et les fonctions séquentielles (toute fonction de flot était potentiellement séquentielle). À l'usage, cela s'est révélé peu satisfaisant. En effet, une fonction séquentielle peut être compilée en une fonction de transition prenant un argument supplémentaire alors qu'une fonction combinatoire n'a pas besoin d'argument supplémentaire. Nous avons finalement opté pour une séparation par typage de ces deux types de fonctions. Le système est décrit dans [19]. Ce système est implanté dans la version 3.

Le calcul d'horloge a été initialement décrit sous forme d'un calcul de types dépendant assez général. Ce calcul a été implanté d'abord dans Lucid Synchrone puis dans le compilateur ReLuC. En observant les planches SCADE, nous avons réalisé qu'un calcul moins expressif mais plus simple serait suffisant pour traiter la plupart des programmes réels. Ceci a conduit à une proposition de calcul d'horloge à la ML [22]. Il est implanté depuis la version 2.

Le rôle de l'analyse de causalité est de rejeter les programmes ne pouvant être ordonnancés statiquement. Cette analyse peut être décrite par un système de types avec rangées [25], ou plus simplement par un système avec des contraintes de sous-typage [52]. L'analyse d'initialisation consiste à rejeter les programmes dont le comportement dépend des valeurs initiales (*nil*) des délais. Une analyse d'initialisation par un système de types avec contraintes de sous-typage a été conçue avec Jean-Louis Colaço et testée sur des applications SCADE de taille réelle [21, 23].

3.3 Définition de systèmes mixtes

Le choix de l'outil de modélisation ou de programmation se pose au tout début de la conception d'un système critique temps réel. Un outil de type schéma/bloc tel que Simulink [57], SCADE/Lustre [29] ou Signal [4] sera mieux adapté à la description d'un système à traitement de données prépondérant (un système de régulation, par exemple). Au contraire, si l'application est à contrôle prépondérant (*drivers*, protocoles), des formalismes plus impératifs ou à base d'automates tels que Stateflow [47], StateCharts [35], les SyncCharts [1] ou Esterel [6] seront plus appropriés. Cependant, une application réelle est souvent un mélange des deux styles. Un exemple typique est celui des commandes de vols où chaque phase du vol est naturellement décrite sous forme

[9] Il suffit, pour cela, d'écrire une fonction calculant la suite $(x_n \& x_{2n})_{n \in \mathbb{N}}$.

de schémas flot de données (décollage, atterrissage, etc.), les transitions d'une loi à l'autre étant spécifiées sous forme d'un automate.

Ceci a conduit à proposer des approches dites multi-paradigmes permettant d'utiliser un langage ou formalisme dédié pour chacun des aspects du système [37, 51, 12, 11, 10] en s'appuyant sur un mécanisme d'édition de liens pour obtenir l'application finale. C'est typiquement ce que fournissent les outils commerciaux tels que Simulink et Stateflow: un diagramme Simulink peut contenir des noeuds spécifiés en Stateflow et qui calculent des flots pouvant contrôler d'autres diagrammes du système. Esterel-Technologies propose des solutions similaires dans sa suite SCADE, en utilisant les SyncCharts comme langage de description des automates. PtolemyII permet également de décrire des systèmes mixtes composés d'équations flot de données et de machines à états. L'approche multi-paradigmes a cependant plusieurs défauts. Elle ne fournit pas une sémantique unique applicable à l'ensemble du système et force à une séparation forte des deux parties du système au tout début de la conception. Satisfaisante lorsque de petits automates gouvernent de grosses lois de commande, elle apparaît limitée sur des systèmes plus équilibrés. Enfin, l'utilisation de générateurs de code différents pour chaque partie du système réduit la possibilité d'obtenir du bon code (lisible et efficace) rendant plus délicate la phase de certification.

Ces observations sont à l'origine d'approches plus intégrées, permettant d'exprimer au sein d'un même langage une programmation flot de données et une programmation du contrôle. C'est l'approche des *Automates de Modes* de Maraninchi et Rémond [44, 45] et de l'ajout de structures de contrôle à un langage flot de données [34, 32, 33]. Ces travaux ont servis de base a la proposition d'extension de Lustre et Lucid Synchrone avec des constructions d'automates [20] et illustrée dans la section précédente. Ces automates permettent d'exprimer de nombreux types de transitions (faibles/fortes, avec réinitialisation éventuelle de l'état but). De plus, cette extension est pleinement conservative dans la mesure où tous les programmes du langage de base sont encore acceptés et gardent la même sémantique. Cela est essentiel pour une intégration dans un outil industriel tel que SCADE. L'idée centrale est de fonder cette extension sur l'utilisation du mécanisme des horloges, en traduisant les constructions impératives dans des programmes flot de données avec horloges. Cette approche se révèle bénéfique à plus d'un titre. Elle force à définir une sémantique précise de l'extension et est d'une grande aide au moment de l'adaptation des analyses statiques du langage de base (typage, calcul d'horloge, initialisation); elle est légère à implanter et permet surtout de réutiliser le générateur de code existant. Les résultats pratiques montrent que le code obtenu est comparable au code écrit à la main ou au code obtenu par une méthode de compilation dédiée [46]. Enfin, les techniques de compilation du flot de données sont maintenant suffisamment comprises pour être acceptées par les autorités de certification. Cette considération a largement motivé l'approche choisie.

4 Conclusion

Ce chapitre a présenté à travers une collection d'exemples l'état actuel des développements autour du langage Lucid Synchrone. Fondé sur le modèle synchrone de Lustre, mais en le reformulant dans le cadre des langages fonctionnels, il offre des mécanismes d'ordre supérieur, de synthèse automatique des types et des horloges et la possibilité de décrire, dans un cadre unifié une programmation flot de données et une programmation par automates.

On voit ici que le langage a joué le rôle pour lequel il avait été défini au départ: un laboratoire d'expérimentations pour les extensions de SCADE/Lustre.

5 Remerciements

Le travail sur Lucid Synchrone s'est développé principalement à l'université Pierre et Marie Curie et a grandement bénéficié des nombreuses discussions avec Thérèse Hardin. Il doit également beaucoup à la collaboration avec Jean-Louis Colaço d'Esterel-Technologies. Nous les en remercions chaleureusement.

References

[1] Charles André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *CESA*, Lille, july 1996. IEEE-SMC. Available at: `www-mips.unice.fr/~andre/synccharts.html`.

[2] E. A. Ashcroft and W. W. Wadge. *Lucid, the data-flow programming language*. A.P.I.C. Studies in Data Processing, Academic Press, 1985.

[3] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[4] A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[5] G. Berry. Real time programming: Special purpose or general purpose languages. *Information Processing*, 89:11–17, 1989.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] Gérard Berry. The esterel v5 language primer, version 5.21 release 2.0. Draft book, 1999.

[8] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming (ICFP)*. ACM, 1998.

[9] Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for Free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.lri.fr/~pouzet`.

[10] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. *Heterogeneous Concurrent Modeling and Design in Java*. Memorandum UCB/ERL M04/27, EECS, University of California, Berkeley, CA USA 94720, July 2004.

[11] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of computer Simulation*, 1994. special issue on Simulation Software Development.

[12] Reinhard Budde, G. Michele Pinna, and Axel Poigné. Coordination of synchronous programs. In *International Conference on Coordination Languages and Models*, number 1594 in Lecture Notes in Computer Science, 1999.

[13] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[14] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating Discrete-Time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 2005. Special Issue on Embedded Software.

[15] Paul Caspi and Marc Pouzet. A Functional Extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 1995. World Scientific.

[16] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[17] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, March 1998. Extended version available as a VERIMAG tech. report no. 97–07 at `www.lri.fr/~pouzet`.

[18] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. *N*-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, South Carolina, USA, January 2006.

[19] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.

[20] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[21] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. In *Synchronous Languages, Applications, and Programming*, volume 65. Electronic Notes in Theoretical Computer Science, 2002.

[22] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[23] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):245–255, August 2004.

[24] The coq proof assistant, 2007. http://coq.inria.fr.

[25] Pascal Cuoq and Marc Pouzet. Modular Causality in a Synchronous Stream Language. In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[26] Conal Elliott. An embedded modeling language approach to interactive 3d and multimedia animation. *IEEE Transactions on Software Engineering*, 25(3):291–308, May-June 1999.

[27] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[28] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[29] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[30] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[31] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[32] Grégoire Hamon. *Calcul d'horloge et Structures de Contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML*. PhD thesis, Université Pierre et Marie Curie, Paris, France, 14 novembre 2002.

[33] Grégoire Hamon. Synchronous Data-flow Pattern Matching. In *Synchronous Languages, Applications, and Programming*. Electronic Notes in Theoretical Computer Science, 2004.

[34] Grégoire Hamon and Marc Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *ACM International conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[35] D. Harel. StateCharts: a Visual Approach to Complex Systems. *Science of Computer Programming*, 8-3:231–275, 1987.

[36] David Harel and Amir Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 477–498. Springer Verlag, 1985.

[37] M. Jourdan, F. Lagnier, P. Raymond, and F. Maraninchi. A multiparadigm language for reactive systems. In *5th IEEE International Conference on Computer Languages*, Toulouse, May 1994. IEEE Computer Society Press.

[38] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[39] Sava Krstic and John Matthews. Semantics of the reFLect Language. In *PPDP*. ACM, 2004.

[40] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.

[41] Edward A. Lee and Pravin Varaiya. *Structure and Interpretation of Signals and Systems*. Addison-Wesley, 2003.

[42] Xavier Leroy. The Objective Caml system release 3.10. Documentation and user's manual. Technical report, INRIA, 2007.

[43] Luc Maranget. Les avertissements du filtrage. In *Actes des Journées Francophones des Langages Applicatifs*. Inria éditions, 2003.

[44] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer verlag.

[45] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, (46):219–254, 2003.

[46] F. Maraninchi, Y. Rémond, and Y. Raoul. Matou : An implementation of mode-automata into dc. In *Compiler Construction*, Berlin (Germany), March 2000. Springer verlag.

[47] The Mathworks. *Stateflow and Stateflow Coder, User's Guide*, release 13sp1 edition, September 2003.

[48] J. Matthews, J. Launchbury, and B. Cook. Specifying microprocessors in hawk. In *International Conference on Computer Languages*. IEEE, 1998.

[49] John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in hydra. In Springer-Verlag, editor, *Functional Programming Languages in Education*, pages 195–214, 1995.

[50] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[51] Axel Poigné and Leszek Holenderski. On the combination of synchronous languages. In W.P.de Roever, editor, *Workshop on Compositionality: The Significant Difference*, volume LNCS 1536, pages 490–514, Malente, September 8-12 1997. Springer Verlag.

[52] Marc Pouzet. *Lucid Synchrone: un langage synchrone d'ordre supérieur*. Paris, France, 14 novembre 2002. Habilitation à diriger les recherches.

[53] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[54] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[55] Mary Sheeran. mufp, a language for vlsi design. In *ACM Conference on LISP and Functional Programming*, pages 104–112, Austin, Texas, 1984.

[56] Walid Taha. Multi-stage programming: Its theory and applications. Technical Report CSE-99-TH-002, Oregon Graduate Institute of Science and Technology, November 1999.

[57] 2007. `www.mathworks.com`.

[58] Jean Vuillemin. On Circuits and Numbers. Technical report, Digital, Paris Research Laboratory, 1993.

[59] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[60] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *International Conference on Programming Language, Design and Implementation (PLDI)*, 2000.