Un Simulateur Synchrone pour Lucid synchrone

Grégoire Hamon & Marc Pouzet

Laboratoire d'Informatique de Paris 6, Case 168, 4 place Jussieu, 75252 Paris Cedex 05, France
`hamon@spi.lip6.fr`, `Marc.Pouzet@lip6.fr`

Résumé

Cet article présente la réalisation d'un outil de simulation graphique pour Lucid synchrone, un langage de flots de données combinant le synchronisme de Lustre et la fonctionnalité de Ml et dédié à la programmation des systèmes réactifs. Ce simulateur est lui même réalisé en Lucid synchrone. L'utilisation du langage nous a offert une approche simple et modulaire de la programmation d'une interface graphique.

1. Introduction

Les langages flots de données synchrones permettent de décrire de manière modulaire des systèmes réactifs, c'est à dire des systèmes réagissant continuement à leur environnement et à la vitesse imposée par celui ci. Toutefois l'activité de mise au point dans de tels langages est souvent rendue délicate par l'aspect dynamique des programmes obtenus.

Nous avons souhaité disposer d'un outil de simulation graphique pour Lucid synchrone [5], un langage flot de données synchrone. Un tel outil doit proposer une interface graphique permettant d'agir sur le déroulement d'un programme et d'observer son comportement. Nous allons voir que ces interfaces sont elles mêmes des systèmes réactifs et que le langage est bien adapté a leur réalisation. Nous montrerons que l'approche flots de données permet de décrire simplement les différents objets graphiques nécessaires à leurs réalisation. De plus, nous verrons que la notion de synchronisme apporte une réponse simple aux différents problèmes de concurrence et d'ordonnancement posés par ces interfaces.

1.1. Lucid Synchrone

Les langages synchrones permettent de décrire les systèmes réactifs de manière parallèle et déterministe. Leur caractère synchrone — les processus partagent une échelle de temps commune — permet une compilation efficace sous forme de machine d'état fini. Ces langages peuvent être de type impératif (e.g, ESTEREL [2]), où à flot de données (e.g, SIGNAL [1] ou LUSTRE [6]).

LUCID SYNCHRONE est un langage né de l'étude des liens entre les langages à flot de données synchrones tels que LUSTRE et les langages fonctionnels de la famille ML [4]. C'est un langage fonctionnel typé de flots synchrone bâti au dessus de ML (précisément OBJECTIVE CAML [8]) dont il reprend la syntaxe et le modèle d'exécution. Un programme manipule des séquences infinies (ou *flots*) comme valeurs de base et adopte le synchronisme de LUSTRE: chaque flot est muni d'une *horloge* indiquant à quels instants le flot est défini. Les opérations sur les flots doivent alors vérifier la cohérence des horloges (on ne peut par exemple faire la somme de deux flots que s'ils ont la même horloge). Cette vérification, appelée *calcul d'horloge* est réalisée statiquement et, à la différence de LUSTRE, les horloges sont inférées à la manière d'un calcul de type. Un programme LUCID SYNCHRONE est constitué d'un ensemble de déclarations de fonctions de flots (appelés *nœuds*) qui seront compilées de manière séparée en machines à états, chaque machine ayant un comportement cyclique. A chaque instant, une "réaction" a lieu: la machine lit une entrée, effectue un calcul simple, modifie sa mémoire interne et produit une sortie. Le programme obtenu est dit "réactif" lorsque le temps de réaction et la mémoire interne sont bornés et que cette borne ne dépend pas des entrées.

1.2. Un simulateur pour Lucid Synchrone

Lors de la mise au point du langage, nous avons très rapidement ressenti le besoin de disposer d'un outil de visualisation et de test permettant d'observer le comportement dynamique d'un nœud. Cet outil serait également très utile dans un but d'apprentissage de la programmation synchrone à flot de données.

Un tel outil de visualisation existe pour le langage LUSTRE [9] et répond très bien à ces objectifs, apprentissage, mise au point et test. Il s'agit d'un générateur de simulateur, produisant une interface graphique pour chaque nœud à simuler. Cette interface permet la saisie en continu des entrées du nœud et calcule et affiche à chaque instant les sorties correspondantes.

En nous fondant sur les idées suivantes :

- L'ajout des effets de bord (e.g, entrée/sorties) à un langage flot de données synchrone est très utile, en particulier pour développer des applications autonomes ou pour observer un programme.

- Un simulateur graphique est un exemple typique de système réactif. En effet, il surveille continuement les périphériques (le clavier, la souris,...) et doit réagir de manière instantanée en produisant les sorties du nœud à simuler et gérer les affichages des objets graphiques.

- Un simulateur graphique est un système parallèle déterministe: il est composé d'un ensemble d'objets graphiques concurrents (e.g, des boutons) se comportant comme des systèmes dynamiques déterministes très simples.

- Enfin, un simulateur est un programme faisant essentiellement des effets de bord (entrées/sorties, affichages,...).

Nous avons choisi d'ajouter les effets de bord au langage et de développer le simulateur directement en LUCID SYNCHRONE. Pour un nœud donné, le système produira automatiquement une interface graphique synchrone permettant de simuler ce nœud. La contribution essentielle de ce travail est de montrer que la programmation à flot de données synchrone se marie bien avec de tels effets et que munis de ces effets, elle est bien adaptée à la réalisation d'interfaces graphiques.

La partie 2 sera consacrée à une introduction brève au langage. Dans la partie 3, nous présenterons les différents choix de conception que nous avons fait. Dans la partie 4, nous détaillerons la réalisation du simulateur graphique.

2. Présentation du langage

Nous allons présenter rapidement les constructions du langage à travers un exemple, et les notions de flots et d'horloge. Considérons la déclaration du *nœud*:

```
let nat h =
  let rec x = 0 -> (pre x) + 1
  in (x, x when h)
```

Cet exemple introduit plusieurs opérateurs:

- le flot infini est la valeur de base du langage. Ainsi, `1` désigne une séquence infinie. De même, les opérateurs scalaires habituels (e.g, `+`) sont appliqués point-à-point.

- le langage reprend les opérateurs temporels de lustre. `->` (ou *followed by*), est un opérateur d'initialisation, qui à l'instant initial rend la valeur de son premier argument, puis rend la valeur de son deuxième argument aux instants suivants. `pre` est un opérateur de décalage. Il retourne une valeur indéfinie à l'instant initial et la valeur précédente de son argument à tout autre instant. `when` permet de sous-échantillonner un flot sur une condition booléenne et permet d'introduire la notion d'horloge. `x when h` retourne une sous séquence de `x` qui est définie aux instants où `h` est vrai. Si `x` et `h` sont sur une même horloge "`cl`", `x when h` a l'horloge "`cl on h`".

Ce programme construit donc le flot `x` des entiers naturels. Le résultat de `nat h` est un flot de paires dont le premier élément est `x` et le second est `x` pour tous les instants où `h` est vrai et n'est pas défini sinon.

`h`	true	true	false	true	false	false	true
`1`	1	1	1	1	1	1	1
`x`	0	1	2	3	4	5	6
`pre x`	nil	0	1	2	3	4	5
`x when h`	0	1	$\perp$	3	$\perp$	$\perp$	6
`nat h`	(0,0)	(1,1)	(2,$\perp$)	(3,3)	(4,$\perp$)	(5,$\perp$)	(6,6)

Le compilateur infère les types et les horloges suivantes:

```
node nat :  bool -> int * int
node nat :: (h:'a) -> 'a * ('a on h)
```

La première ligne donne le type du nœud. La deuxième ligne définit une horloge polymorphe et se lit de la manière suivante: `nat` est une fonction telle que pour toute horloge `a`, si son argument `h` est sur l'horloge `a`, alors sa sortie est sur l'horloge `a * (a on h)` [1].

A partir d'une telle définition, le compilateur produit un programme OBJECTIVE CAML séquentiel et réactif car travaillant sur une mémoire finie (ici, un accummulateur entier) et ne contenant plus de récursion. L'élimination de la récursion est possible car la définition récursive de `x` est causale: la valeur de `x` à l'instant n dépend de la valeur de `x` à l'instant $n-1$.

3. Choix de conception

3.1. Lucid Synchrone simulé en Lucid Synchrone

Le choix du langage dans lequel implanter le simulateur s'est très rapidement tourné vers LUCID SYNCHRONE. Nous avons en effet constaté

[1] Cette horloge est un exemple de type dépendant polymorphe s'écrivant $\forall a.\forall x : a.a * a$ on h.

que le langage était bien adapté à une telle réalisation et qu'il apporte des solutions simples aux différents problèmes posés tant par la simulation que par la description de l'interface graphique.

Nous avons dit qu'un simulateur graphique était un bon exemple de système réactif devant réagir en temps et mémoire bornés. Il est donc raisonnable d'utiliser LUCID SYNCHRONE pour le programmer.

Cette solution présente de nombreux avantages, par rapport notamment au passage dans le langage hôte ou tout autre langage d'usage général (impératif ou fonctionnel). En effet, il est impératif que le simulateur respecte les horloges des nœuds simulés, c'est à dire qu'il doit lire les entrées ou afficher les sorties seulement lorsque leur horloge est vraie. Il sera donc nécessaire de calculer ces horloges. En restant dans le langage, nous laissons le compilateur se charger de ce calcul. Le choix d'un autre langage nous aurait en fait obligé à reprogrammer dans le simulateur une partie du compilateur.

Il nous est également apparu que la programmation flot de données synchrone permettait de décrire de manière modulaire le comportement des différents objets graphiques ainsi que leur mise en parallèle. De plus, la notion d'horloge apporte une réponse simple aux problèmes d'ordonnancement des lectures/écritures induits par les interfaces graphiques, le compilateur se chargeant automatiquement de cet ordonnancement. Enfin, la réactivité du simulateur sera une propriété gratuite s'il est écrit dans le langage lui-même.

Comme nous l'avons déjà dit, ce choix a été rendu possible par l'utilisation d'effets de bord dans le langage. Ceci est obtenu de manière assez naturelle. En effet, les fonctions à effet de bord comme toutes les fonctions scalaires importées du langage hôte OBJECTIVE CAML sont appliquées point-à-point et préservent donc les longueurs: leurs entrées et leurs sorties sont sur la même horloge. Ainsi, une telle fonction n'est appliquée que lorsque son entrée est présente (son horloge est vraie).

3.2. Principe du simulateur

Un simulateur consiste en la mise en parallèle d'un processus de lecture des entrées, du nœud à simuler et d'un processus d'affichage des sorties. Ceci s'écrira (en supposant que `node` soit le nœud à simuler):

```
let node_simul () =
    let input = get_input () in
    let ouput = node input in
    draw_output output
```

Rappelons le caractère flot de données du langage: `input` et `output` désignent respectivement des flots infinis d'entrées et de sorties. A chaque instant, des entrées seront donc lues, les sorties correspondantes calculées par appel du nœud simulé et ces sorties affichées.

Rappelons également son caractère synchrone: le compilateur va inférer que la sortie de `get_input` suit la même horloge que l'entrée de `node` mais aussi que la sortie de `node` est sur la même horloge que l'entrée de `draw_output`. Et tout naturellement, les opérations d'entrées/sorties ne seront effectuées qu'à leur horloge, ce qui correspond au comportement désiré.

Les fonctions `get_input` et `draw_ouput` sont elles-mêmes écrites en Lucid synchrone et consistent en la mise en parallèle de fonctions de lecture et d'écriture de valeurs élémentaires (e.g, un entier, un booléen). Ces dernières correspondront chacune à un objet graphique (e.g, un bouton). Par exemple, un bouton est une fonction de flots recevant en continu des informations venant de la souris (e.g, sa position, son état) et émettant un flot de booléens.

Il est important de pouvoir définir deux comportements très différents pour le simulateur :

Mode pas à pas
: Dans le cas d'un nœud à plusieurs entrées, l'utilisateur demande explicitement le passage à l'instant suivant et donc le calcul des sorties. Il peut donc donner plusieurs entrées à chaque instant.

Mode réactif
: Le simulateur réagit instantanément à la présence d'un événement, tel que la modification d'une entrée.

Ces deux comportements correspondent en fait à deux horloges indiquant quand le simulateur doit calculer et afficher des sorties.

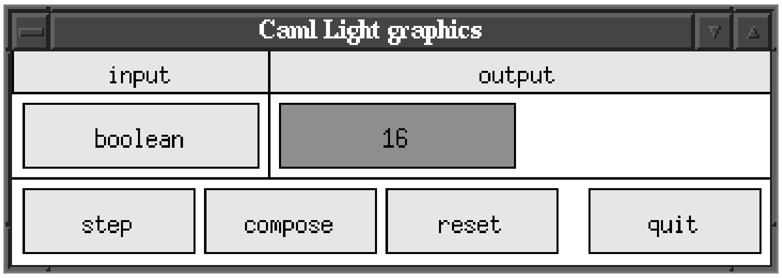
4. Réalisation du simulateur graphique

L'interface graphique elle-même est réalisée dans le langage. Les primitives graphiques élémentaires (affichage, position et état de la souris, etc...) sont importées du langage hôte, mais le comportement dynamique des objets graphiques (e.g, des boutons) est implanté directement dans le langage. Ce simulateur graphique proposera:

- une zone d'entrée pour chaque paramètre,
- une zone d'affichage pour chaque sortie,

- un bouton de pas-à-pas (`step`) permettant de passer à l'instant suivant (et lancer le calcul des sorties en fonction des entrées),
- un bouton de choix du mode de simulation (*pas-à-pas* vs *réactif*); comme dans le simulateur de LUSTRE, ces deux modes sont appelés `compose` et `autostep`,
- un bouton `reset` permettant de ré-initialiser la simulation, c'est à dire de ramener tout les flots à leur instant initial,
- un bouton `quit` permettant de quitter le simulateur.

Sur l'exemple introductif (partie 2), le simulateur aura l'aspect suivant:



Il nous faut donc avant tout disposer de boutons. Nous allons voir que de tels objets se marient bien à une programmation flot de données. Puis nous regarderons comment peuvent être réalisées les entrées/sorties et enfin nous nous intéresserons aux deux modes de fonctionnement ainsi qu'a la ré-initialisation du simulateur.

4.1. Gestion des boutons

Un bouton peut être assimilé à un flot de booléens dont la valeur à chaque instant correspond à son état (*true* = enfoncé, *false* = relevé). Partant de cette constatation, il est assez simple de programmer tous les types de boutons.

Si l'on considère le flot indiquant pour chaque instant si l'on clique sur le bouton ou non, deux types d'évènements peuvent survenir : un front haut (passage de la valeur *false* à la valeur *true*), à l'instant ou l'on enfonce le bouton de la souris et un front bas quand on le relâche. Ces deux événements se programment ainsi:

```
let raising_edge x = x -> not (pre x) & x
let falling_edge x = false -> (pre x) & not x
```

Ceci nous permet de programmer des boutons simples. Nous avons choisi de donner à ces boutons la valeur *true* après un front bas (dès que l'utilisateur relache le bouton). Ces boutons ne conservent cette valeur qu'un instant. Concernant l'affichage, le bouton doit apparaître enfoncé après un front haut et être affiché relevé après un front bas. Ce bouton peut donc être implanté ainsi :

```
let simple_button m_status =
  let bc = button_clic m_status in
  let fh = raising_edge bc in
  let fb = falling_edge bc in
  aff_button (fh when (fh or fb));
  fb
```

Le bouton prend en paramètre le flot des états de la souris, qui indique pour chaque instant les coordonnées de la souris et l'état de ses boutons. Nous utilisons ce paramètre pour calculer le flot `bc` qui indique à chaque instant si l'on clique sur le bouton ou non. Nous calculons le front haut et le front bas de ce flot. La fonction `aff_button` est une fonction à effet de bord importée de Objective Caml. Elle permet d'afficher le bouton, enfoncé si son paramètre est vrai, relevé sinon. Il est très important de noter ici que cet effet n'a lieu que lorsque son horloge est vraie (ici `fh or fb`). On affiche donc le bouton enfoncé sur un front haut (`fh` vaut *true*) et relevé sur un front bas (`fh` vaut *false*). Il ne nous reste plus qu'a retourner la valeur du bouton, qui est *true* lorsqu'il y a un front bas, c'est à dire le flot `fb`.

Remarque 1 (Effets de bord et séquence) On peut remarquer ici l'apparition de la séquence `;`. L'ajout de cette instruction au langage s'est fait naturellement avec l'ajout des effets de bord. En effet une instruction de la forme `let x = e1 in e2` revient, si `x` n'apparaît pas libre dans `e2`, à mettre les deux instructions `e1` et `e2` en séquence. Les deux expressions `e1` et `e2` sont exécutées séquentiellement dans l'instant. On retrouve donc ici la solution classique pour traiter les effets de bord dans les langages fonctionnels stricts. Les définitions `let` et les séquences ; sont éxécutées séquentiellement. Pour les autres constructions, l'ordre d'évaluation n'est pas spécifié.

Un bouton à bascule (`flipflop`) se programme simplement. Il s'agit d'une bascule déclenchée par un front bas. On se dote donc d'une fonction bascule:

```
let flipflop x =
  let rec ff = x -> if x then not (pre ff) else pre ff in ff
```

et le bouton s'écrit directement :

```
let flipflop_button m_status  =
  let bc = button_clic m_status in
  let fh = raising_edge bc in
  let fb = falling_edge bc in
  let out = flipflop fb in
  aff_button (out when (fh or fb));
  out
```

Ce bouton a un fonctionnement similaire au bouton simple. La valeur du bouton bascule sur les front bas et nous affichons cette valeur à chaque front haut et à chaque front bas.

On peut noter que l'approche flot de données nous permet ici de décrire simplement ce que l'on souhaite observer, que cela soit pour la partie dynamique du bouton (sa valeur) ou pour sa partie affichage.

Nous pouvons donc créer les différents boutons de contrôle du simulateur. Les boutons `step`, `reset` et `quit` sont des boutons simples. Le bouton `mode` permettant de sélectionner le mode de simulation est un bouton à bascule. Ils nous suffit ensuite de mettre ces boutons en parallèle. Le code général du simulateur devient:

```
let node_simul () =
    let m_status = get_mouse_status () in

    let step = step_button m_status
    and quit = quit_button m_status
    and breset = reset_button m_status
    and mode = mode_button m_status in

    let input = get_input m_status in
    let output = node (input when step) in
    draw_output output
```

La position de la souris est lue à chaque instant par la fonction `get_mouse_status` qui est importée de OBJECTIVE CAML. Les boutons sont surveillés à chaque instant. Les entrées du nœud à simuler (`node`) sont filtrées par la condition `step`: le calcul des sorties et leur affichage n'a lieu qu'à la demande de l'utilisateur (lorsque celui-ci sélectionne le bouton step). Les flots définis par les boutons `mode` et `reset` ne sont pas utilisés pour l'instant. Le simulateur fonctionne donc pour l'instant en mode `compose`. Le calcul des sorties se fait donc explicitement sur l'horloge définie par le flot `step`.

L'approche présentée ici pour la réalisation de boutons peut permettre de décrire tous les objets graphiques traditionnels, tels que des zones d'entrées, des boutons radio ou des cases à cocher.

4.2. Affichage des sorties

Nous souhaitons afficher un bouton pour chaque sortie. Ces boutons n'ont aucune partie dynamique, ils se contentent d'afficher une valeur. Les fonctions d'affichage en elles-mêmes sont importées de Objective Caml. Le point important est le respect des horloges des sorties. On ne veut afficher une sortie que quand elle est présente, c'est à dire si son horloge est vraie.

C'est ici que l'on gagne vraiment à avoir choisi d'utiliser le langage lui-même pour programmer le simulateur puisqu'il n'y a absolument rien à faire. En effet, la fonction d'affichage des valeurs, fonction à effets de bord, sera évaluée à l'horloge de ses arguments, l'affichage de chaque sortie se fera donc sur l'horloge de cette dernière, et c'est ce que l'on souhaite. L'utilisation d'un autre langage, par exemple le langage hôte, nous aurait obliger à calculer l'horloge de chaque flot de sortie avant de pouvoir faire l'affichage.

4.3. Lecture des entrées

4.3.1. Les différentes zones d'entrée

Il nous faut tout d'abord lire les valeurs en entrée, ceci doit être fait en permanence. Il nous faut ensuite placer les entrées lues sur une horloge compatible avec l'horloge du nœud simulé.

Les lectures en elle-même suivent le même principe que les boutons. Ainsi, à une zone d'entrée de type `t`, on associe un flot de type `t`.

Nous souhaitons pouvoir ré-initialiser les entrées. Nos fonctions d'entrée prendront donc un paramètre booléen indiquant quand cette ré-initialisation doit avoir lieu. Nous utilisons pour cela la primitive `reset/every` du langage qui permet de ramener une fonction de flot à son instant initial.

Exemple 1 Considérons la déclaration suivante:

```
let from n = nat where
  rec nat = n -> pre nat + 1

let res_nat c =
  reset from 0 every c
```

On calcule ici le flot des entiers naturels commençant à `n` mais ce flot est ré-initialisé dès que le paramètre `c` est vrai. Une exécution de ce programme est la suivante :

`c`	false	false	false	true	false	true	true
`res_nat c`	0	1	2	0	1	0	0

Nous allons donc utiliser cette primitive pour ré-initialiser nos zones d'entrées sur réception d'un signal.

- Les entrées booléennes sont des boutons à bascule, prenant un paramètre booléen permettant de les ré-initialiser.

```
let but m_status res =
  let bc = button_clic m_status in
  let fh = raising_edge bc
  and fb = falling_edge bc in
  let out = flipflop fb in
  let o = if res or fb then out else fh
  and b = true -> res or fh or fb in
  o,b

let x_1_button m_status res =
  let o,b = reset but m_status res every res in
  aff_button (o when b);
  o
```

  La fonction intermédiaire `but` calcule les sorties du bouton ainsi que l'horloge à laquelle il doit être affiché. Dans le bouton principal `x_1_button`, `but` est réinitialisé à chaque fois que `res` est vrai.

  Nous avons choisi de ré-initialiser ces entrées après chaque transition (*ie* `step` était vrai à l'instant précédent). Elles doivent également être ré-initialisées après appui sur le bouton reset. La lecture d'une entrée booléenne se fera donc ainsi (on suppose que `breset` est la valeur renvoyée par le bouton reset) :

```
let x_1 =
    x_1_button m_status (false -> pre (step or breset))
```

  on ne ré-initialise bien évidement pas à l'instant initial.

- les autres types d'entrées (*int*, *float*, *string* et *char*) fonctionnent toutes sur le même modèle. Si l'utilisateur a cliqué sur l'entrée, une valeur est lue au clavier, sinon la valeur courante est conservée. Seule la fonction de lecture, importée de OBJECTIVE CAML, différencie les différents types. La définition de ces zones d'entrée est très proche de la définition des boutons booléens. Elle n'est donc pas donnée ici.

4.3.2. Respect de l'horloge du nœud simulé

Nous venons de voir comment nous pouvions lire les valeurs entrées par l'utilisateur. Mais, ces flots doivent pour l'instant être tous sur la même horloge, ce qui n'est pas forcément le cas des paramètres du nœud à simuler. Il nous faut donc filtrer ces flots pour les ramener sur une horloge compatible avec celle du nœud à simuler.

Exemple 2 Soit le nœud suivant :

```
let node (c,x,y) = x when c + y
```

On voit ici que, pour que les deux termes de la somme aient la même horloge, il faut que si `c` est sur l'horloge `'a`, `x` soit sur l'horloge `'a` et `y` sur l'horloge `'a on c`. Effectivement, l'horloge de ce nœud est la suivante :

```
node :: (c:'a) * 'a * ('a on c) -> 'a on c
```

L'horloge de la troisième entrée dépend de la première entrée. Si l'on dispose de flots `x_1`, `x_2`, `x_3` du bon type mais tous sur la même horloge, il nous suffit d'appeler le nœud de la manière suivante:

```
node (x_1, x_2, x_3 when x_1)
```

Nous pouvons donc définir une fonction `node_sim` qui se contente d'appeler le nœud à simuler. Ainsi sur notre exemple :

```
let node_sim x y z = node (x, y, (z when x))
```

l'horloge de ce nœud est alors la suivante :

```
node_sim :: (x:'a) * 'a * 'a -> 'a on x
```

Le nœud ainsi défini prend donc des entrées de même type que le nœud à simuler, mais qui cette fois sont toutes sur la même horloge.

Nous pouvons donc utiliser les horloges du nœud à simuler pour produire un nouveau nœud où toutes les entrées sont sur la même horloge. Les horloges des sorties sont conservées.

4.4. Différents modes de simulation

Le simulateur fonctionne pour l'instant en mode `compose`, seul le bouton `step` déclenche une réaction. Nous souhaitons à présent proposer également un mode `autostep` dans lequel le système réagit dès qu'une entrée (booléenne) est donnée.

L'utilisateur choisit le mode de simulation par l'intermédiaire du bouton de mode.

Reste à implanter le mode `autostep`. La différence entre les deux modes est simplement l'horloge de l'entrée du nœud à simuler. Pour l'instant une transition est déclenchée lorsque `step` est vrai. Nous voulons à présent que cette transition dépende à la fois de `step`, de `mode` et des différentes entrées booléennes. Nous calculons donc l'horloge *réelle* de la simulation, qui prend la valeur `step` quand `mode` est faux (mode `compose`) et la disjonction de `step` et des différentes entrées booléennes lorsque mode est vrai (`autostep`). Par exemple si un nœud a deux entrées booléennes `i_1` et `i_2`, l'horloge réelle sera calculée ainsi:

```
  let real_step = if mode then step or i_2 or i_1 else step
```

Il nous suffit à présent de placer l'appel de la fonction `node_sim` sur l'horloge `real_step` :

```
  let output = node_sim (input when real_step)
```

et de modifier les lectures des entrées booléennes pour qu'elles soient ré-initialisées sur `real_step` et non sur `step`.

On impose en fait ici une horloge au nœud simulé, `real_step`. Il y a donc dans ce simulateur deux horloges. Celle du programme qui est l'horloge de base standard (*i.e*, le flot constant `true`) pour la lecture en continu des entrées et celle de la simulation qui est `realstep`.

Il ne nous reste plus qu'à mettre tous les objets graphiques en parallèle, pour obtenir le simulateur complet:

```
(* synchronisation des entrees *)
let node_sim (x1, ..., xn) = node (y1, ..., yn)

let sim m_status step mode breset =
  (* lecture des entrees *)
  let rec i_1 = x_1_button m_status real_step
  ...
  and i_n = x_n_button m_status real_step
```

```
  (* calcul de l'horloge de simulation *)
  and real_step =
         if mode then step or i_1 or ... i_j else step

  (* calcul des sorties *)
  let (o_1, ..., o_m) =
          node_sim ((i_1, ... ,i_n) when real_step) in

  (* affichage des sorties *)
  clear_out (() when (breset or real_step));
  node_output (o_1, ... o_m)

let sim_main () =
  let m_status = get_mouse_status () in

  let quit = quit_button m_status
  and mode = mode_button m_status
  and step = step_button m_status
  and breset = reset_button m_status in

  reset sim m_status step mode breset every breset
```

On retrouve l'architecture définie initialement. La valeur du bouton `breset` est utilisée pour ré-initialiser tout ce qui concerne la simulation. On peut remarquer que tous les boutons de contrôle de l'interface graphique (`step mode`, `reset` et `quit`) qui indiquent **quand** doit être faite une action particulière, représentent exactement des horloges.

5. Conclusion et perspectives

Nous avons donc montré comment implanter un simulateur pour LUCID SYNCHRONE, lui-même réalisé en LUCID SYNCHRONE. L'implantation présentée ici fonctionne et est intégrée à la distribution du langage [2].

Certaines questions n'ont pas été abordées. Lors de la simulation d'un nœud, si les sorties ne dépendent pas des entrées, elles sont calculées et affichées en permanence (sur l'horloge de base du système) sans qu'aucune entrée ne soit donnée. C'est un cas extrême de sur-échantillonnage qui est dû au fait que le schéma d'horloge du noeud contient plusieurs variables d'horloge. Une solution simple consiste à placer toutes ces variables sur l'horloge `real_step`.

[2] La distribution est disponible par `ftp` anonyme à l'adresse `ftp://spi.lip6.fr` où à l'adresse `http://www-spi.lip6.fr/~pouzet/lucid-synchrone.html`.

De nombreuses extensions et adaptations sont envisageables. En suivant le même principe nous avons ainsi réalisé un simulateur en mode texte et seules les fonctions de lecture et d'écriture ont dû être modifiées [3]. L'intérèt de l'architecture proposée ici est qu'elle s'adapte bien à de telles extensions. Dans le même esprit, il serait intéressant de disposer d'une librairie de composants synchrones, tels que des générateurs de valeurs pour les entrées et différentes forme d'affichage (e.g, sous forme de chronogramme) pour les sorties.

L'idée d'une programmation à flot de données des interfaces graphiques n'est pas nouvelle et a été largement abordée dans les langages fonctionnels paresseux (e.g, les fudgets [3] de Haskell [7]). L'intérêt d'une programmation flot de données synchrone est de conserver le caractère réactif du noeud simulé. On peut d'ailleurs constater que le simulateur se comporte ici plus comme un "observateur" qui regarde fonctionner le programme, que comme un simulateur.

Enfin, nous souhaitions profiter de la mise au point de ce simulateur pour expérimenter l'utilisation d'effets de bord dans un langage flot de données synchrone. Cette expérience nous a permis de constater que la programmation flot de données synchrone munie de ces effets semblait bien adaptée à la réalisation d'interfaces graphiques.

Enfin, l'interface de LUCID SYNCHRONE avec OBJECTIVE CAML s'est avérée très confortable en permettant de programmer d'une manière uniforme, la partie dynamique et la partie scalaire du système.

Remerciements

Nous remercions Thérèse Hardin et Christian Queinnec pour leurs relectures de versions préliminaires de ce travail.

Bibliographie

[1] 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.

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

[3] M. Carlsson and T. Hallgren. Fudgets – a graphical user interface in a lazy functional language. In *Conference on Functional Programming*

[3] Ce simulateur est lui aussi disponible dans la distribution.

Languages and Computer Architecture (FPCA), pages 321–330. ACM Press, June 1993.

[4] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Penn., May 1996.

[5] P. Caspi and M. Pouzet. Lucid synchrone: une extension fonctionnelle de lustre. In *Journées Francophones des Langages Applicatifs (JFLA)*, Morzine-Avoriaz, Février 1999. INRIA.

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

[7] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), 1990.

[8] Xavier Leroy. The Objective Caml system release 1.07. Documentation and user's manual. Technical report, INRIA, 1998.

[9] Pascal Raymond. Xsimlus, a graphical simulator for Lustre programs. Available in the distribution of Lustre. Web: `http://www-verimag.imag.fr`.