Systemes Synchrones
MPRI – cours 2.23-1 2006-2007

Examen
8 décembre 2006

Les exercices sont indépendants. L'énoncé est composé de 3 pages. L'examen dure 2 heures. Les notes de cours sont autorisés.

1 Exercice

1. Une fonction f : $\texttt{bool}^\omega \to \texttt{bool}^\omega$ est causale si elle est monotone pour l'odre préfixe (i.e., $x \leq y \Rightarrow f(x) \leq f(y)$). Donner un exemple de fonction de suites à valeurs booléennes ($\texttt{bool}^\omega \to \texttt{bool}^\omega$) causale mais pouvant dépendre d'un passé non borné de son entrée.
2. Ecrire (en Lustre, Esterel ou sous forme d'automates de Mealy hiérarchiques) un programme qui vérifie que deux signaux booléens `a` et `b` alternent, i.e., il n'est pas possible d'avoir deux occurrences de `a` sans avoir eu une occurrence de `b` (et réciproquement) [1]. Ce programme produit le flot égal à vrai (ou émet le signal `ok`) tant que la propriété est vérifiée.

2 Compilation en circuits

Dans cet exercice, on se propose d'étudier l'ajout de structures de contrôle à un petit langage data-flow de type Lustre. Ce langage sera considéré comme un noyau de base. L'objectif est d'étendre ce langage avec des constructions impératives (dans l'esprit d'Esterel) et de montrer comment celles-ci peuvent être traduites vers le noyau. Nous utiliserons une technique proche de celle utilisée dans la compilation en circuit d'Esterel.

Le noyau de base est défini par la grammaire suivante :

$$\begin{array}{lcl} D & ::= & \{x_1 = e_1; ...; x_n = e_n\} \\ e & ::= & e + e \mid v \mid \texttt{true} \mid \texttt{false} \mid e \,\&\, e \mid e \;\texttt{or}\; e \mid \texttt{not}\; e \\ & & \mid \texttt{pre}\; x \mid e \;\texttt{->}\; e \mid \texttt{if}\; e \;\texttt{then}\; e \;\texttt{else}\; e \end{array}$$

D désigne un système d'équations. On supposera que les x_i sont distincts deux-à-deux. v désigne un flot entier constant. `true` et `false` désignent des flots constants (pour tout $n \geq 0$, $\texttt{true}_n = t$ et $\texttt{false}_n = f$). `pre` x est le retard de Lustre en imposant qu'il ne s'applique qu'à une variable ($(\texttt{pre}\; x)_{n+1} = x_n$ pour $n > 0$). Pour éviter les problèmes d'initialisation, on supposera que la valeur initiale de `pre` x vaut -1 si x est un flot d'entiers et la valeur fausse f si x est un flot de booléens. x `->` y est l'opération d'initialisation de Lustre $(x \texttt{ -> } y)_0 = x_0$ et $(x \texttt{ -> } y)_n = y_n$ pour $n > 0$. La sémantique de la conditionnelle de Lustre est définie par : $(\texttt{if}\; ec \;\texttt{then}\; et \;\texttt{else}\; ef)_n = et_n$ si $ec_n = t$ et $\texttt{if}\; ec \;\texttt{then}\; et \;\texttt{else}\; ef)_n = ef_n$ si $ec_n = f$.

1. Donner une définition sous forme d'un système d'équations produisant une valeur *ok* pour les fonctions suivantes :
   (a) `until`(x) produit un flot initialement faux et qui passe à vrai dès que le flot x a été vrai dans le passé strict. Une fois la valeur vraie émise, celle-ci est maintenue infiniment.
   (b) `unless`(x) produit un flot vrai dès que x est vrai. La valeur retournée est fausse sinon. Une fois la valeur vraie émise, celle-ci est maintenue infiniment.
   (c) `init`(x) est vrai tant que x est faux et devient vrai (pour toujours) dès que x a été vrai dans le passé strict. On remarquera que `init(true)` = `true -> false`.
2. On se propose d'ajouter une structure de contrôle correspondant à une condition d'activation dans les schémas data-flow. Le langage des équations est étendu avec la construction suivante :

$$D \quad ::= \quad \texttt{activate}\; D \;\texttt{when}\; e \mid ...$$

[1] Cette propriété permet de caractériser des systèmes "quasi-synchrones" correspondant à un fonctionnement où les deux ne s'éloignent pas trop l'un de l'autre.

Le principe de cette construction est de n'executer le corps D que si la condition booléenne e est vraie. Sinon, les flots calculés dans D conservent leur valeur précédemment calculée. Ainsi, le programme suivant produit la suite `cpt = -1 -1 42 43 43 43 44 45 45 45 46 47 ...`

```
{ activate
    { cpt = 42 -> pre cpt + 1}
  when cond;
cond = false -> (pre (false -> not (pre cond)))
```

Ce programme est-il équivalent au suivant ? Pourquoi ?

```
{ cpt = if cond then 42 -> pre cpt + 1 else pre cpt;
  cond = false -> (pre (false -> not (pre cond))) }
```

Sinon, proposer une formulation équivalente n'utilisant pas la condition d'activation. *Indication :* L'opération d'initialisation x `->` y apparaissant sous une condition d'activation c ne doit changer d'état que lorsque c a été vraie.

3. Proposer une méthode de traduction du noyau étendu avec cette construction vers le noyau de base. Pour cela, vous pourrez définir une fonction $Trad_c(D)$ prenant en paramêtre une condition booléenne c, une définition D du noyau étendu et produisant un ensemble d'équations $\{x_1 = e_1; ...; x_n = e_n\}$ du noyau de base. c sera la condition d'activation portée par le contexte. Elle sera utilisée dans la compilation des opérations apparaissant dans D.
4. En supposant que D ne contienne pas de boucle de causalité (au sens de Lustre), la construction d'activation `activate` D `when` e doit-elle vérifier des conditions supplémentaires afin que sa traduction ne contienne pas de boucle de causalité ?
5. La condition d'activation peut être généralisée en une forme binaire. Le langage d'équations est étendu ainsi :

$$D \quad ::= \quad \texttt{when}\ e\ \texttt{do}\ D\ \texttt{else}\ D \mid ...$$

On supposera, pour simplifier, que toute variable x est définie dans les deux branches. Ainsi, le programme suivant :

```
{ when cond then
    { cpt = 42 -> pre cpt - 1}
  else
    { cpt = 45 -> pre cpt + 1};
cond = false -> (pre (false -> pre (false -> not (pre cond))))
```

calcule la suite `cpt = 45 46 47 42 41 40 48 49 50 39 38 37 ...`

On peut donc remarquer que `pre cpt` désigne une mémoire locale mise à jour seulement lorsque le code dans lequel elle se trouve est actif. Autrement dit, les deux occurrences de `pre cpt` désigne des mémoires différentes.

Proposer une formulation équivalente au programme précédent mais n'utilisant pas la condition d'activation binaire. Vous pourrez utiliser la condition d'activation unaire si vous le souhaitez. On ne s'attachera pas a l'efficacité de la traduction.

6. Proposer une méthode de traduction de cette construction vers le noyau de base. Vous pourrez, si besoin, introduire des variables fraiches lors de cette traduction.
7. On introduit deux nouvelles constructions permettant d'exprimer une mise en séquence et d'attente (dans l'esprit des opérations de préemption d'Esterel). Le langage d'équations est enrichi ainsi :

$$D \quad ::= \quad \texttt{do}\ D\ \texttt{until}\ e\ \texttt{then}\ D \mid \texttt{do}\ D\ \texttt{unless}\ e\ \texttt{then}\ D \mid ...$$

`do` D_1 `until` e `then` D_2 est une construction de préemption faible. D_1 est exécuté jusqu'à ce que la condition booléenne e soit vraie (instant inclus). Puis, D_2 est exécuté pour toujours. `do` D_1 `unless` e `then` D_2 est une construction de préemption forte. D_1 est exécuté à moins que e ne soit vrai. Dans ce cas, D_2 est alors exécuté infiniment. Ainsi, le programme suivant :

```
do { x = 0 -> pre x + 1 } until (x = 5) then { x = 10 }
```

produit la suite `x = 0 1 2 3 4 5 10 10 10 10 ...` Le programme suivant :

```
do { x = 0 -> pre x + 1 } unless cond then { x = 10 };
cond = false -> pre(false -> true)
```

produit la suite `x = 0 1 10 10 10 ...`

Proposer un schéma de traduction de ces deux constructions en utilisant la condition d'activation binaire `when` . `do` . `else` ..

8. Ces constructions doivent elles vérifier des conditions particulières pour que le programme traduit ne contienne pas de boucle de causalité ?
9. On ajoute maintenant une opération permettant de réinitialiser un comportement lorsqu'une condition booléenne est vraie.

$$D \quad ::= \quad \texttt{reset}\ D\ \texttt{every}\ e \mid ...$$

Ainsi, le programme :

```
reset
  { cpt = 0 -> pre cpt + 1 }
every
  (false -> pre cpt = 5)
```

produit la séquence `cpt = 0 1 2 3 4 5 0 1 2 3 4 5 ...`

L'opération de réinitialisation consiste à faire repartir les flots définis dans le corps à leur valeur initiale. Autrement dit, réinitialiser un calcul avec une condition booléenne r revient à remplacer toute opération d'initialisation x `->` y apparaissant dans D par `if true ->` r `then` x `else` y.

Ainsi, l'exemple ci-dessus produit le même flot `cpt` que le programme suivant :

```
{ cpt = if true -> res then 0 else pre cpt + 1;
  res = false -> pre cpt = 5 }
```

Proposer une méthode de traduction du langage étendu avec la construction `reset` D `every` e vers le noyau de base.

10. (Supplément) On ajoute enfin les deux constructions suivantes :

$$D \quad ::= \quad \texttt{wabort}\ D\ \texttt{when}\ e \mid \texttt{loop}\ D \mid D\ \texttt{then}\ D$$

La construction `wabort` D `when` e se comporte comme l'équation D et se termine lorsque e est vrai. La terminaison est représentée par un flot special noté `end` valant vrai lorsque le programme se termine. La construction D_1 `then` D_2 exécute l'équation D_1 jusqu'à ce que D_1 se termine puis exécute l'équation D_2. La construction `loop` D est un raccouci pour D `then loop` D.

Montrer comment traduire ces nouvelles constructions en utilisant les constructions définies précédemment.