2ème année du MPRI
Université Paris VII

Rapport de lecture
Chantal Keller

A clocked denotational semantics for Lucid-Synchrone in Coq

Sylvain Boulmé *et Grégoire* Hamon

Professeurs: Marc Pouzet
Jean Vuillemin

15 décembre 2008

Introduction

Les langages de programmation synchrones sont des langages de haut niveau servant à décrire des systèmes réactifs. Ils doivent vérifier que les programmes écrits respectent certaines conditions, comme par exemple le synchronisme : des données entrent et sortent du système selon une horloge globale.

Étant donné l'utilisation de ces langages synchrones, par exemple pour gérer les commandes d'un avion de ligne, il est important de pouvoir vérifier formellement ces propriétés sur les programmes. Pour cela, un plongement du langage de programmation synchrone choisi dans un assistant de preuve peut permettre de pouvoir ensuite écrire une spécification des programmes, et de prouver des propriétés dessus.

Dans leur article [1] et leur rapport technique [2], Sylvain Boulmé et Grégoire Hamon ont choisi de plonger Lucid-Synchrone dans Coq. Il s'agit d'un langage de programmation synchrone fonctionnel, qui se prête donc très bien à un plongement dans un assistant de preuve, souvent fonctionnel également par construction.

Ce plongement est un plongement direct qui respecte une idée fondamentale de Lucid-Synchrone : le calcul d'horloges. Chaque signal se voit attribuer une horloge indexée sur l'horloge globale qui indique s'il est présent ou non ; c'est un système de types à part entière. Cela permet d'assurer par exemple qu'un programme bien typé (donc avec des horloges cohérentes) ne peut planter. Ce typage a la particularité d'être décidable et inférable. Dans le plongement en Coq, on garde les horloges comme paramètre des flots leur garantissant de bonnes propriétés.

Sylvain Boulmé et Grégoire Hamon ont parfaitement appuyé leur raisonnement sur une implantation en Coq (version 6) de leurs travaux. Elle montre par de nombreux exemples qu'il est assez facile d'écrire les spécifications d'un programme Lucid-Synchrone dans leur plongement Coq.

Dans une première partie, nous verrons quelles sont les structures de données de base qui permettent un plongement direct. Ensuite, nous nous intéresserons à la définition des opérateurs Lucid-Synchrone en Coq, et verrons combien les possibilités de Coq permettent d'avoir une syntaxe proche de celle de Lucid-Synchrone. La troisième partie sera consacrée à la construction de flux et de fonctions récursives à l'aide d'opérateurs récursifs simples. Enfin, nous détaillerons la traduction de l'implantation Coq de Sylvain Boulmé et Grégoire Hamon dans la version actuelle de Coq.

1 Structures de données

1.1 Présentation

Lucid-Synchrone est un langage de programmation data-flow synchrone. Ainsi, ses types de données de base sont les signaux, (ou flux, ou flots), représentés par des listes infinies (d'entiers, de booléens...). Ces signaux sont paramétrés par une horloge, qui est une liste infinie d'entiers, et qui désigne à chaque instant si un élément du signal est présent ou non.

Comme nous réalisons un plongement direct de Lucid-Synchrone en Coq, une bonne manière de représenter les horloges est d'utiliser les listes infinies de Coq, implantées par la bibliothèque `Streams`. Ce sont des constructions co-inductives. Une horloge est donc une liste infinie de booléens, la valeur `true` représentant un "tic" de cet horloge, et la valeur `false` un moment où le signal n'est pas défini.

Ainsi, un signal est une liste infinie paramétrée par une horloge, et dont les éléments sont :
- absents lorsque l'horloge vaut `false` ;
- présents lorsque l'horloge vaut `vrai`, avec soit une valeur, soit `Fail`.

`Fail` est une valeur particulière, qui permet de désigner les valeurs qui n'ont pas encore été initialisées. En effet, cela peut se produire, soit lorsqu'on souhaite introduire un délai, soit pour définir un signal de manière récursive : dans ce cas, on doit le définir pas à pas, et ce qui n'est pas encore défini à un instant où l'horloge est vraie porte la valeur `Fail`.

Le terme utilisé est celui de *liste échantillonnée*, pour désigner le fait qu'elle peut contenir des valeurs non initialisées :

```
Inductive samplElt (A: Set) : bool -> Set :=
  | None: samplElt A false
  | Any: A -> samplElt A true
  | Fail: samplElt A true

CoInductive samplStr (A: Set) : clock -> Set :=
  | sp_cons: forall (c: clock),
    samplElt A (hd c) -> samplStr A (tl c) -> samplStr A c.
```

Naturellement, la valeur `Fail` sert seulement à la construction des signaux, mais elle ne doit plus apparaître dans un signal final. Un tel signal (n'ayant aucune valeur `Fail`) est dit *bien formé* :

```
Definition is_no_Fail (b: bool) (s: samplElt A b) : Prop :=
  match s with
    | Fail => False
    | _ => True
  end.

CoInductive sp_wf : forall (c: clock), (samplStr A c) -> Prop :=
  sp_wf_proof : forall (c: clock) (s: samplStr A c),
    is_no_Fail (sp_hd s) -> sp_wf (sp_tl s) -> sp_wf s.
```

1.2 Égalité

L'égalité sur les signaux est une égalité **extensionnelle**, c'est-à-dire que deux signaux sont égaux si et seulement si leurs têtes sont les mêmes et leurs queues sont égales (cela correspond à une bisimulation).

Détaillons ce que signifient que "leurs têtes sont les mêmes". En Coq, on ne peut comparer deux termes que si leurs types sont convertibles (au sens de la β-égalité). Mais dans notre cas, on a besoin de pouvoir tester l'égalité de termes qui n'ont pas des types convertibles, mais seulement dont on peut prouver l'égalité des types. C'est pourquoi on utilise une égalité dépendante :

```
x ==<< T >> y
```

signifie que `x` a pour type `T a`, `y` a pour type `T b` et on peut prouver `a=b`.

L'égalité de deux signaux est donc ainsi définie en Coq :

```
CoInductive sp_eq :
  forall (c1 c2: clock), samplStr A c1 -> samplStr A c2 -> Prop :=
  | sp_eq_proof :
  forall (c1 c2: clock) (s1: samplStr A c1) (s2: samplStr A c2),
    ((sp_hd s1) ==<< samplElt A >>(sp_hd s2)) ->
    sp_eq (sp_tl s1) (sp_tl s2) -> sp_eq s1 s2.
```

Cette égalité entre signaux est bien une relation d'équivalence. Elle est également compatible avec une certaine notion de coercition : si `c1` et `c2` sont deux horloges bisimulables, alors on peut transformer un signal paramétré par `c1` en un signal paramétré par `c2` et égal au premier (au sens de l'égalité décrite ci-dessus).

1.3 Implantation

La notation `_ ==<< _ >> _` pour désigner l'égalité dépendante, ainsi que quelques lemmes utiles concernant cette égalité, sont définis dans le fichier `EqD.v`.

Les définitions des signaux, de leur égalité, de la notion de coercition se trouvent dans le fichier `sampled_streams.v`. On dispose alors de nombreuses définitions et de nombreux lemmes, dont voici le schéma principal :

- on peut définir une relation d'ordre entre les horloges : `clock_inf c1 c2` si et seulement si, à chaque fois que `c1` vaut `true`, alors `c2` également. Intuitivement, cela correspond à regarder quand un signal est "plus défini" qu'un autre ;
- étant donné un signal `s`, on peut définir le $n^{\text{ème}}$ élément de `s`, ou encore le signal commençant au $n^{\text{ème}}$ élément de `s`. Ces définitions ont de "bonnes propriétés" de commutation (par exemple, si `s'` commence au $m^{\text{ème}}$ élément de `s`, alors le $n^{\text{ème}}$ élément de `s'` est le $(m+n)^{\text{ème}}$ élément de `s` ...) ;
- un signal est bien formé s'il ne contient pas la valeur `Fail`, mais pour arriver à prouver des propriétés sur les signaux, on a besoin de définir qu'un signal ne contient pas la valeur `Fail` jusqu'au rang n (`loc_nfstwf`) ;
- en combinant les trois point ci-dessus, on peut définir une relation d'ordre sur les signaux : `sp_inf n s1 s2` si et seulement si, pour chacun des n premiers éléments de $s1$, soit son horloge est fausse (il est absent), soit sont horloge est vraie, et dans ce cas, celle de l'élément correspondant de $s2$ est vraie, et ils sont égaux (au sens dépendant). Le cas échéant, si $s2$ est bien formé jusqu'au rang n, alors $s1$ également.

1.4 Conclusion

On dispose d'une structure de données permettant de représenter des signaux synchrones. Il faut maintenant définir des opérateurs correspondant à ceux de Lucid-Synchrone, pour retrouver l'expressivité de ce langage.

Toutes les propriétés succinctement décrites ci-dessus vont s'avérer très utiles pour obtenir des informations sur les signaux définis avec la syntaxe de Lucid-Synchrone. Notamment, on pourra prouver qu'un signal est bien formé.

2 Syntaxe de Lucid-Synchrone

2.1 Présentation

Lucid-Synchrone dispose d'un petit nombre d'opérateurs qui lui confèrent une grande expressivité pour décrire de nombreux signaux récursifs et leurs interactions. Le plongement direct des signaux Lucid-Synchrone dans les listes infinies de Coq permet de plonger également les opérateurs de manière simple et intuitive. De plus, la souplesse de Coq permet de disposer de notations agréables à utiliser.

Il y a trois types d'opérateurs :
- ceux qui génèrent un flux sortant directement à partir du flux entrant ;
- ceux qui introduisent un délai dans le signal entrant ;
- ceux qui calculent un échantillonnage du signal entrant.

Pour chaque cas, nous allons donner un exemple l'illustrant.

2.2 Premier cas : exemple des signaux constants

Un signal constant est la donnée d'une horloge `c` et d'un élément `a`. Lorsque `c` est vraie, le signal correspondant est `a` ; lorsque `c` est fausse, le signal est `None` (absent).

La syntaxe Coq est :

```
Variable A: Set.
Variable a: A.

Definition elt_const (b: bool) : samplElt A b :=
  if b return samplElt A b then Any a else None A.

CoFixpoint sp_const (c:clock) : samplStr A c :=
  sp_cons _ (elt_const (hd c)) (sp_const (tl c)).
```

On peut par exemple montrer qu'un tel signal est bien formé.

En Lucid-Synchrone, 3 représente par exemple le signal entier constamment égal à 3 et défini sur l'horloge de base courante. En Coq, on ne pourra pas le noter 3 car cela entrerait en conflit avec les autres structures de données prédéfinies ; mais on peut quand même utiliser une notation plus agréable que `sp_const nat 3 c`, en posant la déclaration suivante :

```
Notation " ’‘’ c ’‘’ " := (sp_const c _) (at level 100).
```

Le signal constant égal à 3 pourra donc se noter `‘3‘`.

2.3 Deuxième cas : exemple de *followed by*

En Lucid-Synchrone, l'opérateur `fby` est une macro utilisant les opérateurs plus atomiques `pre` et `->`. En Coq, nous allons procéder exactement de la même façon. Mais tout d'abord, il nous faut définir une fonction qui ajoute un élément en tête d'un flot :

```
Variable A: Set.

CoFixpoint sp_delay (c: clock) : samplStr A c -> samplElt A true -> samplStr A c :=
```

```
  match c as c0 return samplStr A c0 -> samplElt A true -> samplStr A c0 with
    | Cons hc tc => if hc as b return
         samplStr A (Cons b tc) -> samplElt A true -> samplStr A (Cons b tc)
         then (fun x a => sp_cons (Cons true tc) a (sp_delay (sp_tl x) (sp_hd x)))
         else (fun x a => sp_cons (Cons false tc) (None A) (sp_delay (sp_tl x) a))
  end.
```

Lorsqu'on ajoute un élément dans un flot, il doit nécessairement être sur le premier tic d'horloge, et il faut décaler tous les autres éléments sur le tic suivant. Il faut donc, lorsque l'horloge est fausse, mettre l'élément `None` et garder en mémoire l'élément courant.

En utilisant cette fonction, `pre` consiste uniquement à ajouter l'élément `Fail` en tête de flot :

```
Definition sp_pre (c: clock) (s: samplStr A c) : samplStr A c :=
  sp_delay s (Fail A).
```

La flèche `->` modifie le premier élément d'un flot :

```
CoFixpoint sp_arrow (c: clock) (x y: samplStr A c) : samplStr A c :=
  sp_cons _ (sp_hd x)
              (if (hd c)
               then (sp_tl y)
               else (sp_arrow (sp_tl x) (sp_tl y))).
```

Enfin, on peut définir `fby` :

```
Definition sp_fby c x y := sp_arrow (c := c) x (sp_pre y).
```

Là encore, on peut utiliser une notation plus proche du langage Lucid-Synchrone grâce aux facilités de Coq :

```
Infix "fby" := sp_fby (at level 100, right associativity).
```

qui définit la constante `fby` comme un opérateur binaire infixe associatif à droite.

Si le flux de départ est bien formé, le flux obtenu en appliquant `fby` est également bien formé.

2.4 Troisième cas : exemple de *when*

L'opérateur `when` consiste à obtenir un sous-échantillonnage d'un signal, en l'indexant sur une horloge plus petite (au sens du premier point de la partie 1.3) que la sienne. Il faut donc d'abord définir quelle est cette horloge plus petite : c'est ce qui se nomme en Lucid-Synchrone `on`. Définissons cet opérateur pour un instant :

```
Definition elt_on (b:bool) (o: samplElt bool b) : bool :=
  match o with
    | None => false
    | Any x => x
    | Fail => true
  end.
```

Si l'élément n'est pas `Fail`, il est retourné, mais s'il s'agit de `Fail`, on renvoie `true` : l'erreur doit se diffuser.

`when` utilise cette fonction en copiant son entrée lorsque l'horloge est vrai, et en l'oubliant lorsqu'elle est fausse :

```
Variable A: Set.

Definition elt_when (b: bool) (o: samplElt bool b) :
         samplElt A b -> samplElt A (elt_on o) :=
  match o as o0 in samplElt _ b0 return samplElt A b0 -> samplElt A (elt_on o0) with
    | None => fun x => None _
    | Any b0 => fun (x: samplElt A true) =>
             if b0 as b0' return samplElt A b0' then x else None _
    | Fail => fun x => Fail _
  end.

CoFixpoint sp_when (c: clock) (lc: samplStr bool c) (x: samplStr A c) :
           samplStr A (sp_on lc) :=
  sp_cons (sp_on lc) (elt_when (sp_hd lc) (sp_hd x))
                          (sp_when (sp_tl lc) (sp_tl x)).
```

Là encore, on peut mettre une notation proche de celle de Lucid-Synchrone :

```
Notation swp_when := (fun lc s => sp_when s lc).
Infix "when" := swp_when (at level 100, right associativity).
```

(remarquons qu'il faut d'abord inverser les arguments de `sp_when`, puisqu'ils n'ont pas été mis dans l'ordre de ceux de Lucid-Synchrone... (voir partie 4.1)).

Si le flux de départ est bien formé, le flux obtenu en appliquant `when` est également bien formé.

2.5 Implantation

Ces opérateurs sont implantés dans le fichier `lucid_ops.v`, ainsi que d'autres opérateurs utiles :

- pour les opérateurs générant un flux sortant directement à partir du flux entrant :
  - `ext`, un opérateur qui prend pour arguments un flux de fonctions et un flux d'arguments, et qui renvoie l'application de chacune des fonctions à chacun des arguments ;
  - `not`, obtenu en appliquant `ext` au flux constant contenant la négation booléenne ;
  - `if`, obtenu en appliquant `ext` au flux constant contenant la condition booléenne ;
- pour les opérateurs calculant un échantillonnage du signal entrant :
  - `merge`, qui fait la fusion de deux signaux indexés sur des horloges complémentaires ;

Pour tous ces opérateurs, on peut montrer des lemmes de préservation de la bonne formation des flux.

Il est également montré qu'on peut éliminer l'opérateur `pre` pour les fonctions sur les signaux qui sont sans mémoire, c'est-à-dire les fonctions soit qui génèrent un flux sortant directement à partir du flux entrant, soit qui calculent un échantillonnage du signal entrant (toutes celles qui n'induisent pas de délai). Cela permet de montrer plus facilement que ces

fonctions induisent des signaux bien formés, puisqu'on ne rajoute pas arbitrairement une valeur `Fail`.

2.6 Conclusion

On a maintenant défini tous les opérateurs de Lucid-Synchrone, et on peut donc complètement simuler ce langage... à condition que l'on dispose d'un moyen de définir des flux récursifs.

3 Constructions récursives et exemples

3.1 Présentation

Lucid-Synchrone dispose de deux notions de constructions récursives : les flux récursifs et les fonctions récursives (fonctions qui prennent des flux comme entrées et sorties, puisque ce sont les types de données primitifs).

On ne peut pas traduire directement ces conditions dans Coq, car Lucid-Synchrone permet d'écrire des programmes qui ne terminent pas alors que Coq ne le permet bien sûr pas. Cependant, on peut décrire des opérateurs de récursion qui permettent de simuler en partie les constructions de Lucid-Synchrone. Ces opérateurs de récursion peuvent être vus comme des cas particuliers d'un opérateur générique.

3.2 Opérateur générique

Cet opérateur, nommé `rec1`, permet de construire toutes les fonctions récursives ayant pour type :

```
forall (x: B), samplStr A (C x)
```

où A et B sont des ensembles **quelconques** et C est une fonction qui, à tout élément de B, associe une horloge. `rec1` procède en calculant un plus petit point fixe d'une fonction, et pour se faire, en utilisant des approximations successives (c'est-à-dire qu'à une étape i, on a l'assurance que le flux que l'on construit est bien formé jusqu'au rang i). Cela correspond à l'intuition qui est derrière les signaux de Lucid-Synchrone.

Cet opérateur est particulièrement puissant, puisque selon l'ensemble B, on peut construire toutes sortes de fonctions récursives (mais qui terminent) ainsi que des flux récursifs. C'est à ce cas que nous allons particulièrement nous intéresser.

3.3 Flux récursifs

Dans le cas particulier où B est le type `unit`, c'est-à-dire un type qui ne contient qu'un seul constructeur (`tt : unit`), `rec1` permet de construire des flux récursifs. C'est cela qui définit l'opérateur de récursion pour les flux :

```
Definition rec0 :=
  rec1 (fun (x: unit) => c) (fun (f: unit-> samplStr A c) (x: unit) => F0 (f x)) tt.
```

Une fois encore, on utilise une notation plus simple d'utilisation :

```
Notation "'<' c '>rec0' " := (rec0 (c := c)) (at level 100).
```

Cet opérateur permet de définir tous les exemples classiques de flux récursifs, comme celui des entiers naturels :

```
Definition Nat c :=
  <c>rec0 (fun N => `0` fby (ext (`S`) N)).
```

ou la suite de Fibonacci :

```
Definition Fib c :=
  <c>rec0 (fun fib => `(1)` fby (ext (ext (`plus`) fib) (`0` fby fib))).
```

Pour définir des flux corrects (dans Coq, mais aussi dans Lucid-Synchrone), l'argument auquel on applique `rec0` doit vérifier deux conditions :

- les fonctions doivent, à chaque instant, prendre un élément échantillonné en argument, effectuer un calcul, et renvoyer un élément échantillonné. Cela assure que l'ensemble du système est bien synchrone ;
- les fonctions agissant sur les flux doivent augmenter strictement le rang jusqu'auquel les flux sont bien formés. Cela permet d'assurer la terminaison de l'opérateur `rec0`.

Ces conditions sont bien sûrs issues de conditions similaires pour `rec1`.

Enfin, notons que `rec0` est suffisamment puissant pour définir des flux mutuellement récursifs, qui sont souvent nécessaires en Lucid-Synchrone ; cependant, son utilisation devient alors assez technique.

3.4 Implantation

Le fichier `sp_rec.v` implante `rec1` et `rec0`, ainsi que des lemmes montrant que les préconditions décrites ci-dessus permettent, si elles sont vérifiées, d'assurer que les opérateurs de récursion génèrent des flux bien formés.

Le fichier `examples.v` donne de nombreux exemples d'utilisation de `rec0` et `rec1`. Il montre qu'un utilisateur qui souhaiterait prouver des propriétés sur un programme Lucid-Synchrone pourrait assez facilement utiliser la syntaxe, assez intuitive, décrite dans les parties 2 et 3. Pour pouvoir ensuite faire des preuves, l'implantation fournit un raffinement de la tactique `auto` de Coq qui permet d'automatiser les preuves sur de petits exemples. En revanche, on constate que sur des exemples un peu plus difficiles, `auto` ne suffit plus et il faut alors avoir une bonne connaissance des lemmes et de Coq pour pouvoir effectuer les preuves de propriétés de programmes.

4 Traduction de Coq v.6 vers Coq v.8

4.1 Présentation

Une partie importante de mon travail a consisté à porter le code originel, qui est écrit pour la version 6.3.1 de Coq, pour la version 8.1 (version stable actuelle). Ce travail est terminé, les fichiers `EqD.v`, `sampled_streams.v`, `lucid_ops.v`, `sp_rec.v` et `examples.v` ont été **entièrement** traduits.

Ces fichiers traduits sont fournis avec ce document, ainsi qu'un `Makefile` permettant de les compiler.

Dans cette traduction, j'ai essayé de **respecter le plus fidèlement possible** le code originel : même lorsqu'une commande de Coq a été profondément remodelée (comme `Notation`), j'ai ajouté les paramètres adéquats permettant de garder le sens du code de départ. Une seule différence de taille est à noter : certains arguments qui étaient *implicites* dans la version 6 deviennent *explicites* dans la version 8 (mais cela est dû à un remodelage de Coq). Il est également important de noter que la syntaxe `_ ==<< _ >> _` pour les égalités dépendantes n'est pas celle de départ (qui était `_ ==< _ > _`), car elle devenait incompatible avec la nouvelle version de la bibliothèque standard.

4.2 Détails

La syntaxe de Coq a énormément évolué depuis la version 6. En effet, aujourd'hui, les mots-clés `forall` et `fun` permettent d'avoir une écriture des types et des termes beaucoup plus naturelle lorsqu'on est habitué à la programmation fonctionnelle. De fait, il a fallu reprendre toute la syntaxe (mais cela peut facilement s'automatiser).

Concernant les preuves, la très grande majorité d'entre elles restaient valides. Certaines ont dû cependant être modifiées, notamment celles faisant intervenir la fonction `eq_dep_ind_l` : la version 6 était capable de résoudre le problème d'unification (du second ordre) posé par l'application de cette fonction, mais pas la version 8. Plus généralement, ces problèmes d'unification ont fait que certains arguments qui pouvaient être implicites dans la version 6 ont dû être explicités dans la version 8. Ces modifications sur les preuves sont beaucoup plus difficiles à automatiser ! (justement, elles doivent pallier un défaut d'automatisation de la part de Coq...)

4.3 Conclusion

Le traduction ne m'a donc pas présenté de difficulté particulière, mais m'a permis de :

- lire en détail l'implantation Coq de Lucid-Synchrone, et ainsi bien la comprendre et bien comprendre pourquoi elle fonctionne ;
- apprendre la syntaxe de Coq v.6 (ce qui peut m'être utile, car cet exemple n'est pas le seul a être écrit dans cette ancienne version !)
- apprendre des constructions de Coq qui m'étaient jusque là inconnues (l'élimination dépendante des constructeurs, la tactique `lapply`, la commande `Notation`...) ;

De plus, l'actualisation d'un tel code peut permettre, sinon son utilisation encore aujourd'hui, du moins de conserver les intérêts de cette étude assez colossale sur la sémantique dénotationnelle de Lucid-Synchrone.

Conclusion

Coq est un outil tout à fait désigné pour créer les outils permettant de spécifier des programmes Lucid-Synchrone, et donc d'effectuer des preuves formelles dessus. En effet, on obtient une syntaxe très proche de celle de Lucid-Synchrone, et il est donc très aisé, même sans avoir connaissance de tous les outils, de pouvoir écrire la spécification d'un programme Lucid-Synchrone dans ce formalisme.

Outre cet aspect pratique, l'implantation effective dans Coq donne une sémantique dénotationnelle de Lucid-Synchrone très concise et claire, ainsi que la preuve formelle de bonne formation de certains programmes. De nombreux lemmes ont été prouvés qui permettent d'être immédiatement appliqués à de véritables programmes, même assez complexes.

Cependant, le travail mérite d'être développé dans certaines directions :

- si la spécification de programmes complexes est aisée, les preuves que l'on souhaite réaliser dessus restent d'un accès difficile si on n'a pas une connaissance complète du code Coq. Cet inconvénient peut se voir amélioré par le développement de Coq et de la tactique `auto` ;
- les auteurs ont fait certains choix, comme l'utilisation de la valeur `Fail`, qui sont pratiques à utiliser, mais ne reflètent pas exactement le mode opératoire de Lucid-Synchrone.

La traduction de l'implantation originelle dans une version de Coq plus récente peut permettre de relancer l'étude de ces questions, dont la portée s'étend sur Coq mais également sur la sémantique des langages de programmation synchrone.

Références

[1] G. Boulmé, S. Hamon. *LNCS*, chapter Certifying synchrony for free. Springer Verlag, 2001.

[2] G. Boulmé, S. Hamon. A clocked denotational semantics for lucid-synchrone in coq. Technical report, LIP6, 2001.