☺

Cours MALG & MOVEX

MOVEX2

Modélisation synchrone

Dominique Méry
Telecom Nancy, Université de Lorraine

Année universitaire 2025-2026 (30 avril 2026 6:52 A.M.)

Summary

1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes
10. Contrats
11. Lustre in Kind2
    - Back to Lustre Programs
    - Basic definitions
    - Temporal Operators
    - Declarative semantics
    - Composite Types
12. Contracts for LUSTRE in Kind2
13. Propriétés des programmes LUSTRE

Summary

1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes
10. Contrats
11. Lustre in Kind2
    - Back to Lustre Programs
    - Basic definitions
    - Temporal Operators
    - Declarative semantics
    - Composite Types
12. Contracts for LUSTRE in Kind2
13. Propriétés des programmes LUSTRE

Summary

1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE

Summary

1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes
10. Contrats
11. Lustre in Kind2
    - Back to Lustre Programs
    - Basic definitions
    - Temporal Operators
    - Declarative semantics
    - Composite Types
12. Contracts for LUSTRE in Kind2
13. Propriétés des programmes LUSTRE

Systèmes réactifs : contexte

- Systèmes interagissant en continu avec leur environnement
- Domaines :
  - Contrôle de procédés industriels
  - Contrôle de véhicukes autonomes
  - Traitement du signal
  - Protocoles de communication
  - Interfaces homme-machine
  - Dispositifs médicaux
- Exigence clé : **temps de réponse très court**

Observation 1 : Parallélisme

- Interaction continue et simultanée avec l'environnement
- Architecture souvent distribuée :
  - performance
  - tolérance aux fautes
  - modularité
- Conception naturelle en **modules parallèles coopérants**
- ...

Observation 1 : Parallélisme

- Interaction continue et simultanée avec l'environnement
- Architecture souvent distribuée :
  - performance
  - tolérance aux fautes
  - modularité
- Conception naturelle en **modules parallèles coopérants**
- ... avec une implémentation centralisée

Observations 2 et 3 : Temps et sûreté

Contraintes temporelles

- Fréquences d'entrée
- Temps de réponse entrée/sortie
- Contraintes imposées par l'environnement
- Doivent être :
  - spécifiées
  - vérifiées
  - garanties

Sûreté (Dependability)

- Systèmes critiques (avionique, nucléaire, médicaux, autonomes, ...)
- Une erreur peut avoir des conséquences graves
- Nécessité de :
  - méthodes dites formelles
  - vérification rigoureuse

Summary

1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes
10. Contrats
11. Lustre in Kind2
    - Back to Lustre Programs
    - Basic definitions
    - Temporal Operators
    - Declarative semantics
    - Composite Types
12. Contracts for LUSTRE in Kind2
13. Propriétés des programmes LUSTRE

Hypothèse de synchronisme

- Paradigme pour systèmes réactifs et temps réel
- Exemples : Lustre, Esterel, Signal

Hypothèse de synchronisme

- Paradigme pour systèmes réactifs et temps réel
- Exemples : Lustre, Esterel, Signal
- Basé sur une **sémantique formelle rigoureuse**
- **Temps discret global**
- **Calculs instantanés (abstraction)** : **machine infiniment rapide**
- Lecture des entrées et production des sorties à chaque instant

Modèle des flots

- Une variable est un flot :

$$x = (x_0, x_1, x_2, \dots)$$

- Les programmes sont des fonctions :

$$F : Streams^n \rightarrow Streams^m$$

- Calcul point par point

Opérateurs temporels

- `pre` : valeur précédente

$$(pre\ x)_n = x_{n-1}$$

- `->` : initialisation

$$x = a \rightarrow b$$

Causalité

- Dépendance uniquement du présent ou du passé
- Interdiction des cycles instantanés
- Analyse statique de causalité

Déterminisme

- Une même entrée $\Rightarrow$ (produit) la même sortie
- Pas de concurrence non déterministe
- Comportement prévisible

Compositionnalité

- Systèmes construits par assemblage de blocs
- Tout bloc est une fonction synchrone
- Preuves locales $\Rightarrow$ preuve globale

Traduction en modèles formels

- Automates finis
- Circuits séquentiels
- Équations booléennes

Vérification formelle

- Model checking (Kind2)
- Preuve de propriétés :
  - sûreté
  - absence de blocage
  - contraintes temporelles

Comparaison des langages

Langage	Paradigme	Point fort
Lustre	flot de données	analyse
Esterel	impératif	contrôle
Signal	relationnel	multi-horloges

Propriétés théoriques

- Déterminisme
- Réactivité
- Causalité
- Sémantique mathématique
- Équivalence avec automates

Applications

- Systèmes embarqués critiques
- Avionique
- Ferroviaire
- Systèmes cyber-physiques

Conclusion

- Modèle rigoureux et vérifiable
- Adapté aux systèmes critiques
- Forte base théorique

Summary

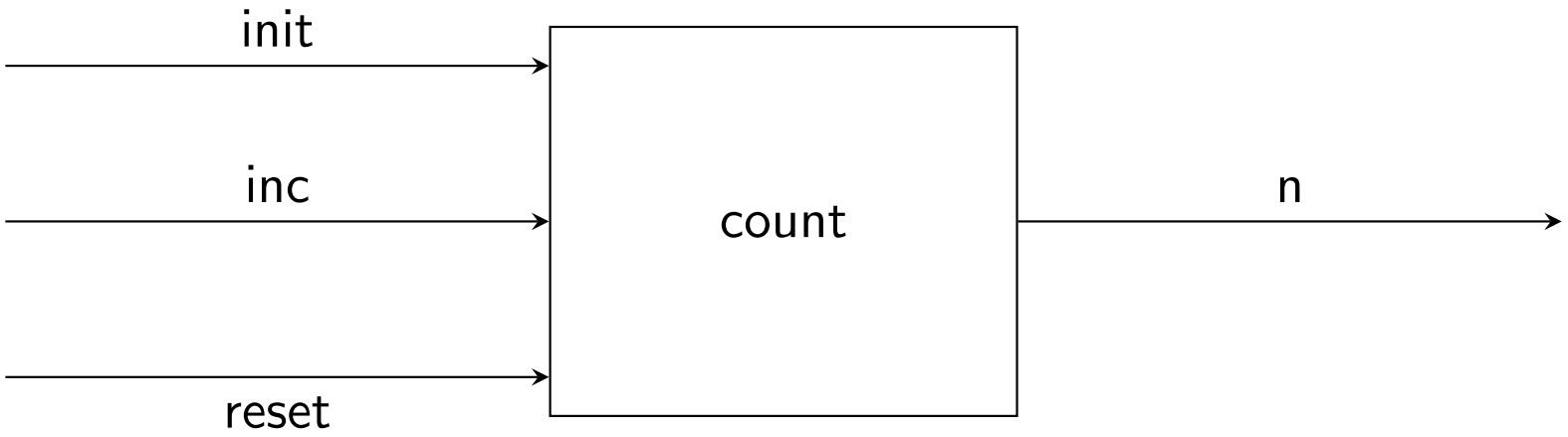
1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Le langage LUSTRE

- LUSTRE est un langage synchrone à flôts de données.
- LUSTRE est la base de SCADE un environnement de dévelopement utilisé dans les véhicules de transport (avionique).
- LUSTRE a la possibilité d'une représentation graphique (Paul Caspi et Nicolas Halbwachs, 1984)

Le langage LUSTRE

- LUSTRE est un langage synchrone à flôts de données.
- LUSTRE est la base de SCADE un environnement de dévelopement utilisé dans les véhicules de transport (avionique).
- LUSTRE a la possibilité d'une représentation graphique (Paul Caspi et Nicolas Halbwachs, 1984)



Le langage LUSTRE

- LUSTRE est un langage synchrone à flôts de données.
- LUSTRE est la base de SCADE un environnement de dévelopement utilisé dans les véhicules de transport (avionique).
- LUSTRE a la possibilité d'une représentation graphique (Paul Caspi et Nicolas Halbwachs, 1984)

Le langage LUSTRE

- LUSTRE est un langage synchrone à flôts de données.
- LUSTRE est la base de SCADE un environnement de dévelopement utilisé dans les véhicules de transport (avionique).
- LUSTRE a la possibilité d'une représentation graphique (Paul Caspi et Nicolas Halbwachs, 1984)

(Flôt de données)

Listing 2 – count.lus

```
node COUNT (init,incr:int;reset:bool) returns (n: int);
let
    n = init  -->
        (if reset then init else  pre(n) +incr);
tel;
```

Le langage LUSTRE

- LUSTRE est un langage synchrone à flôts de données.
- LUSTRE est la base de SCADE un environnement de dévelopement utilisé dans les véhicules de transport (avionique).
- LUSTRE a la possibilité d'une représentation graphique.

Le langage LUSTRE

- LUSTRE est un langage synchrone à flôts de données.
- LUSTRE est la base de SCADE un environnement de dévelopement utilisé dans les véhicules de transport (avionique).
- LUSTRE a la possibilité d'une représentation graphique.

(Flôt de données)

Listing 4 – dataflow1.lus

```
-- count
node count(x,y:int)  returns (r: int);
let

  r = 5*(x+y);

tel
```

Langage à flôt de données

- production : Un flôt d'entrées produit un flôt (flux) de sorties :

$$flot \in input^N \rightarrow outpout^N$$
$$flot(i_0, i_1, \ldots i_n) = (o_0, o_1, \ldots o_n)$$

- réaction :

$$flot \in input \times state \rightarrow output \times state$$
$$flot(in, s_t) = (out, s_{t+1})$$

Le code engendré correspond à la fonction de cycle.

Forme générale d'un module LUSTRE

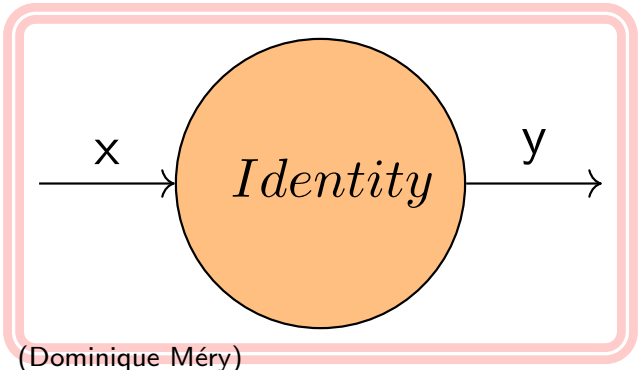
node $f(x_1 : \alpha_1, \ldots, x_n : \alpha_n)$ returns $(y_1 : \beta_1, ..., y_m : \beta_m)$
var $z_1 : \gamma_1, ..., z_k : \gamma_k;$
let
$z_1 = \delta_1; ...; z_k = \delta_k;$
$y_1 = \epsilon_1; ...; y_m = \epsilon_m;$
assert $P_1; \ldots;$ assert $P_l;$
tel

Exemple du nœud Identity

Node Identity

node $I(x : int)$ returns $(y : int)$
let
$y = x$
assert $x = y$;
tel

- Copie de la stream x vers la stream y.
- $x = (x_0, x_1, x_2, \ldots, x_i, \ldots)$
- recopie
- $y = (y_0, y_1, y_2, \ldots, y_i, \ldots)$



Programmes LUSTRE

- Un programme LUSTRE est une liste de modules appelés des nœuds.
- Tous les nœuds fonctionnent de manière synchrone.
- Les communications sont réalisées via les inputs et les outputs.
- Les équations doivent avoir des solutions et ne sont pas des affectations.

streams

- Toutes les variables, constantes et expressions sont des streams.
- Les opérations classiques sont étendues sur les streams :
  - $x = (0, 1, 2, 3, 4, \ldots)$
  - $y = (1, 2, 3, 4, \ldots)$
  - $x{+}y = (1, 3, 5, 7, \ldots)$
- Chaque stream correspond à une horloge.

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Semantical Concepts for Reactive Programming

4	4	4	...	4	...
x	x_0	x_1	...	x_n	...
y	y_0	y_1	...	$ytest_n$	...
x+y	x_0+y_0	x_1+y_1	...	x_n+y_n	...

x	x_0	x_1	...	x_n	...
pre x	NIL	x_0	...	x_{n-1}	...

x	x_0	x_1	...	x_n	...
y	y_0	y_1	...	y_n	...
x->y	x_0	y_1	...	y_n	...

```
nat = 0 -> (1 + pre nat)
```

h	**true**	**false**	**true**	**true**	**false**
x	x_0	x_1	...	x_n	...
x when h	x_0	—	x_2	x_3	—

Programmes LUSTRE

- **Un programme LUSTRE s'appelle un nœud ou NODE.**
- **Un programme LUSTRE dénote une suite infinie de valeurs comme suit :** $(x_0\ x_1\ x_2\ \ldots)$
- **Deux opérateurs sur les programmes :**
  - $\mathbf{pre}$
  - $\longrightarrow$
- $\forall n \geq 0. CUP_{n+1} = CUP_n + 1$ **s'écrira** $CUP = 0 \longrightarrow (1 + \mathbf{pre}(CUP))$
- **et produira la suite** $(0\ 1\ 2\ \ldots)$**.**
- $FIB = 0 \longrightarrow 1 \longrightarrow (\mathbf{pre}(FIB) + \mathbf{pre}(\mathbf{pre}(FIB)))$

Programmes LUSTRE

- $$\begin{array}{l} node \quad EDGE(X : bool) \quad returns \quad (Y : bool) \\ let \\ \quad Y = false \rightarrow \ X \ and \ not \ pre(X) \\ tel \end{array}$$

  désigne la suite $(false, x_1 \wedge \neg x_0, x_2 \wedge \neg x_1, \ ...)$

- **Counter**

  $C = 0 \longrightarrow \mathbf{pre}(C)+1$ **renvoie la suite des naturels**

  $C = 0 \longrightarrow if \ X \ then \ \mathbf{pre}(C)+1 \ else \ \mathbf{pre}(C)$

  compte le nombre d'occurrences de X **qui sont vraies.**

  On ignore la valeur initiale

  $PC = 0 \longrightarrow \mathbf{pre}(C)$

  $C = if \ X \ then \ PC+1 \ else \ PC$

Compteur

- $$\begin{array}{l} node COUNTER(init, incr : int; X, reset : bool) returns (C : int) \\ let \\ \quad PC = init -> \ pre\ C \\ \quad C = if\ reset\ then\ init \\ \quad\quad else\ if\ X\ then\ (PC + incr) \\ \quad\quad else PC; \\ tel \end{array}$$

- $odds = COUNTER(0, 2, true, true -> false)$ **définit les entiers impairs.**

Constructions LUSTRE

- **Deux opérateurs sur les programmes :**
  - **pre**
  - $\longrightarrow$

  T
- $X\ when\ B$
- $current\ X$
- $assert$
  - $assert\ not\ (x\ and\ y\ )$
  - $assert\ (true- > not(x\ and\ pre(x)))$

Compteur

```
node COUNTER(init,incr:int; X,reset:bool) returns (C:int)
let
   PC=init->  pre C
   C = if reset then init
       else if X then (PC+incr)
       else PC;
tel
```

- $odds = COUNTER(0, 2, true, true- > false)$ définit les entiers impairs.

Constructions et opérateurs LUSTRE

- **Deux opérateurs sur les programmes :**
  - **pre**
  - $\longrightarrow$
- $X\ when\ B$ **: filtre de** X **quand la valeur de** B **est vraie.**
- $current\ X$ **: interpolation de** X
- $and, not, or, xor, \ldots$ **sont des opérateurs booléens sur les streams.**
- $assert$
  - $assert\ not\ (x\ and\ y\ )$
  - $assert\ (true- > not(x\ and\ pre(x)))$
- **réutilisation de nœuds**

```
node FALLING_EDGE(X:bool) returns (Y:bool)
let
   Y= EDGE(not X);
tel
```

Intégration

- **Soit f une fonction du temps à valeurs réelles et on souhaite l'intégrer selon la méthode des trapèzes.**
- **Deux valeurs sont reçues par le programme $F_n = f(x_n)$ et $x_{n+1} = x_n + STEP_{n+1}$**
- **Calcul de Y : $Y_{n+1} = Y_n + (F_n + F_{n+1}) \cdot STEPn + 1/2$**
- **La valeur de Y est une donnée**

```
node integration(F,STEP,init:real) returns (Y:real)
let
   Y= init -> pre(Y)+ ((F + pre(F))*STEP)/2.0;
tel
```

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Tools for LUSTRE

- Development Tools for Critical Reactive Systems using the Synchronous Approach The Verimag Reactive Tool Box
- Compilation of LUSTRE program into C programs.

Multi-engine SMT-based automatic model checker for synchronous reactive systems

- Automatic analysis of LUSTRE porograms
- SMT-based tools
- k-induction

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Exemple de programme LUSTRE (1)

(Sommation)

Listing 5 – sum.lus

```
-- the current value of Sum(X) is the sum of all values of X up to now
node sum (X: int) returns (S: int);
let
   S = X -> X + (pre S);
tel

--        0   1   2   3   4   5 ...
--  ------------------------------
--  X =   5,  3, -1,  2,  7,  8, ...
--  S =   5,  8,  7,  9, 16, 24, ...
```

Exemple de programme LUSTRE (2)

(Fibonacci)

Listing 6 – fibo.lus

```
-- This node produces the Fibonacci series: 1,1,2,3,5,8,13,21,...

node fibo(_:bool) returns(Fib: int);
let
   Fib = 1 -> pre (1 -> Fib + pre Fib);
tel
```

Exemple de programme LUSTRE (3)

(Opérations)

Listing 7 – operations.lus

```
-- operations over nodes
node op (X: int;Y:int) returns (S,P,POLY,POLY2:int;TEST:bool);
let

   S =  X + Y;
   P = X*Y;
   POLY = X*X + 2*X*Y + Y*Y;
   POLY2 = S*S;
   TEST = (POLY = POLY2);
   check TEST;

tel
```

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Horloges

- Toutes les expressions sont des streams.
- Les opérateurs d'horloge modifient la temporalité des streams et le résultat est stream :
  - `pre` s pour toute stream s.
  - $s1 -> s2$ s1 *suivi de* s2.
  - $s1$ `when` $s2$ *échantillonnage*
  - `current` s pour toute stream s.

Sémantique des horloges

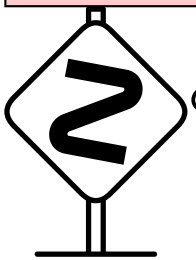
- $[\![\sigma]\!] \stackrel{def}{=} (\sigma_0, \sigma_1, \ldots)$
- $[\![\texttt{pre}(\sigma]\!]) \stackrel{def}{=} (\bot, \sigma_0, \sigma_1, \ldots)$
- $[\![\sigma - > \tau]\!]) \stackrel{def}{=} (\sigma_0, \tau_1, \tau_2, \ldots)$
- $[\![\sigma \texttt{ when } \varphi]\!]) \stackrel{def}{=} (\sigma_{t_0}, \tau_{t_1}, \tau_{t_2}, \ldots)$ où la suite des t_i est la suite des instants validant φ dans la suite σ.
- $[\![\texttt{current}(\sigma \texttt{ when } \varphi]\!])) \stackrel{def}{=}$
  $(\bot, \ldots, \bot, \sigma_{t_0}, \sigma_{t_0}, \sigma_{t_0}, \sigma_{t_1}, \sigma_{t_1}, \sigma_{t_1}, \sigma_{t_1}, \sigma_{t_1}, \sigma_{t_2}, \ldots)$

Exemple d'horloges

(Horloge)

Listing 8 – clock1.lus

```
node clock1(b: bool) returns (y: int);
var n:int;
var e:bool;
var f:int;
let
  n = 0 -> pre(n)+1;
  e = true ->  not pre(e);
  f = current ( n when e);
  y = current ( n when e) div 2;
tel
```



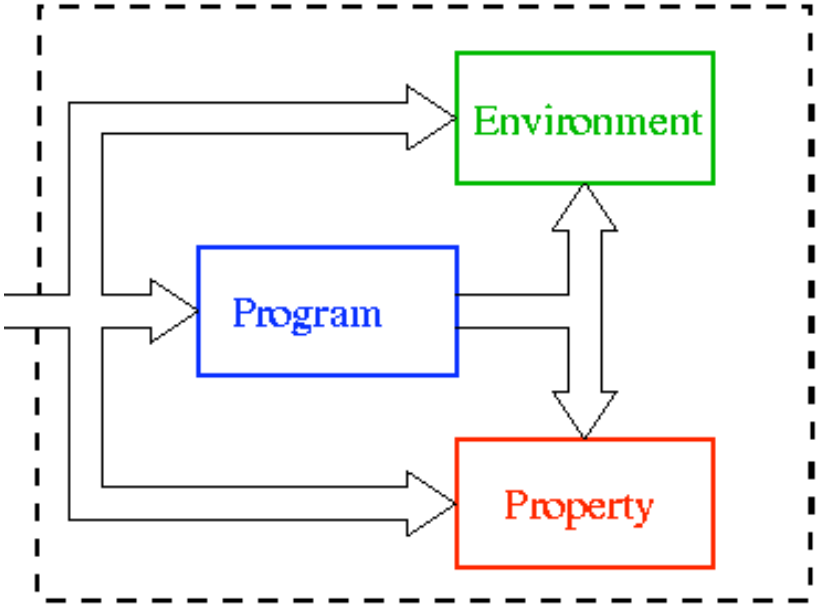
`current` n'est pas supporté.

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Vérification par observateurs

- **Description de la propriété à vérifier et les hypothèses de l'environnement**
- **Un observateur d'une propriété de sûreté est un programme qui utilise comme entrée les entrées-sorties du programme à vérifier et décide en émettant un signal à chaque instant si la propriété est violée ou non**



Programmer un switch

- **Transformer un signal en niveau par un swith qui est utilisé comme suit :**
  - **deux signaux possibles en entrée set et reset**
  - **une valeur initiale initial**
  - **toute occurence de set fait passer le niveau à true**
  - **toute occurence de reset fait passer le niveau à false**
  - **quand aucun des signaux n'apparaît, le niveau ne change pas**
- **un signal est modélisé comme un booléen :**

```
node SWITCH1(set,reset,initial: bool) returns (level:bool)
let
   level = initial -> if set the true
                      else if reset then false
                      else pre(level);
tel
```

- **Problème : ce programme ne modélise pas un switch à un bouton :**

```
state =  SWITCH1(change,change,true)
```

Programmer un switch à un bouton

- **Transformer un signal en niveau par un swith qui est utilisé comme suit :**
  - **deux signaux possibles en entrée set et reset**
  - **une valeur initiale initial**
  - **toute occurence de set fait passer le niveau à true**
  - **toute occurence de reset fait passer le niveau à false**
  - **quand aucun des signaux n'apparaît, le niveau ne change pas**
-
```
node SWITCH(set,reset,initial: bool) returns (level:bool)
let
   level = initial -> if set and not pre(level)  then true
                      else if reset then false
                      else pre(level);
tel
```
- **Vérification :**

```
node verification(set,reset,initial: bool) returns (ok:bool)
let
   level = SWITCH(set,reset,initial);
   level1 = SWITCH(set,reset,initial);
   ok = (level = level1);
   assert  not(set  and reset)
tel
```

Vérification de power2

(power2 avec vérification)

Listing 9 – power2prop.lus

```
-- power2(_) contains all the square  numbers in increasing order
-- where 0^2 = 0
--       (n+1)^2 = n^2 +2*n + 1
node power2(_ : bool) returns (P: int);
var W: int;
var P2:int;
var PROP:bool;
var N:int;
let
  -- all the natural even numbers
  W = 1 -> (pre W) + 2;
  P = 0 -> (pre P) + (pre W);
  N = 0 -> (pre N)+1;
  P2 = N*N;
  PROP  = P2 = P;
  check PROP;

tel
```

Application de la k-induction : `kind2 --enable BMC --enable IND`

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Propriétés des programmes LUSTRE

()

Listing 10 – kindsession2/power2.lus

```
-- power2(_) contains all the square  numbers in increasing order
-- where 0^2 = 0
--       (n+1)^2 = n^2 +2*n + 1
node power2() returns (P: int);
var W: int;
let
  -- all the natural even numbers
  W = 1 -> (pre W) + 2;
  P = 0 -> (pre P) + (pre W);
tel
```

Propriétés des programmes LUSTRE

()

Listing 12 – kindsession2/power2.lus

```
-- power2(_) contains all the square  numbers in increasing order
-- where 0^2 = 0
--       (n+1)^2 = n^2 +2*n + 1
node power2() returns (P: int);
var W: int;
let
  -- all the natural even numbers
  W = 1 -> (pre W) + 2;
  P = 0 -> (pre P) + (pre W);
tel
```

(`--enable BMC --enable IND`)

Listing 13 – kindsession2/power2prop.lus

```
-- power2(_) contains all the square  numbers in increasing order
-- where 0^2 = 0
--       (n+1)^2 = n^2 +2*n + 1
include "power2.lus"
node power2prop() returns (P: int);
var P2,PP2,N:int;
var PROP:bool;
let
  PP2 = power2();
  N = 0 -> (pre N)+1;
  P2 = N*N;
  PROP  = P2 = PP2;
  check PROP;
tel
```

()

Listing 14 – kindsession2/greycounter.lus

```
node greycounter (reset: bool) returns (out: bool);
var a, b: bool;
let
  a = false -> (not reset and not pre b);
  b = false -> (not reset and pre a);
  out = a and b;

tel
```

()

Listing 15 – kindsession2/intcounter.lus

```
node intcounter (reset: bool; const max: int) returns (out: bool);
var t: int;
let
  t = 0 -> if reset or pre t = max then 0 else pre t + 1;
  out = t = 2;
tel
```

Propriétés des programmes LUSTRE

(Valid property)

Listing 16 – kindsession2/ex1.lus

```
/* include   */
include "greycounter.lus"

/* include   */
include "intcounter.lus"


node top (reset: bool) returns (OK, OK2: bool);
var b, d: bool;
let
  b = greycounter(reset);
  d = intcounter(reset, 3);
  OK = b = d;
  --%PROPERTY OK;
tel
```

Propriétés des programmes LUSTRE

(Invalid property)

Listing 17 – kindsession2/ex2.lus

```
/* include   */
include "greycounter.lus"

/* include   */
include "intcounter.lus"


node top (reset: bool) returns (OK, OK2: bool);
var b, d: bool;
let
  b = greycounter(reset);
  d = intcounter(reset, 3);
  OK = b = d;
  OK2 = not d;
  --  %PROPERTY OK;
  --%PROPERTY OK2;
tel
```

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Treillis complet

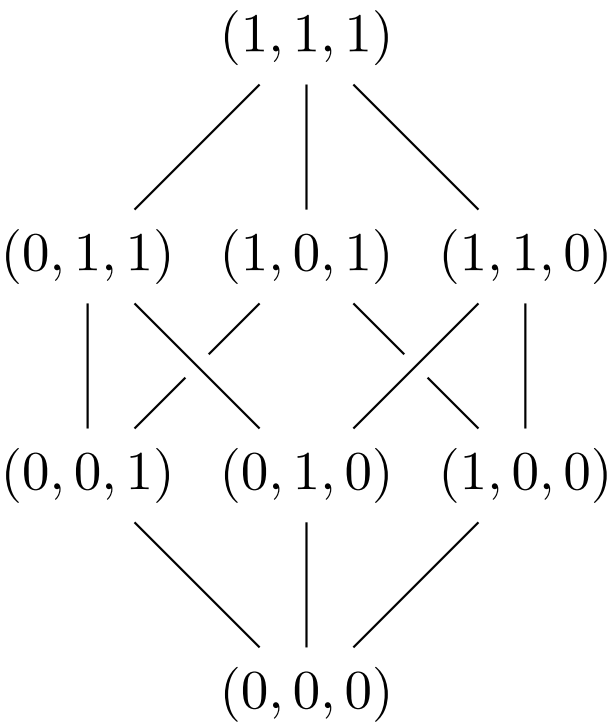
⊠ Definition

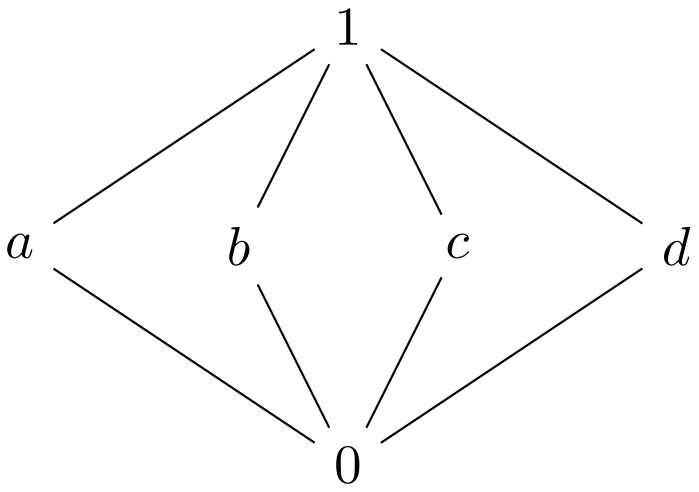
Un treillis complet $(L, \sqsubseteq, \bot, \sqcup, \top, \sqcap)$ est un treillis satisfaisant les propriétés suivantes :

1. Pour toute partie A de L, il existe une borne supérieure notée $\sqcup A$.
2. Pour toute partie A de L, il existe une borne inférieure notée $\sqcap A$.

- Un treillis complet $(L, \sqsubseteq,)$ peut être défini par les éléments suivants $(L, \sqsubseteq, \bot, \sqcup, \top, \sqcap)$
- Un treillis est une structure munie d'un ordre partiel et telle que deux éléments ont une borne supérieure et une inférieure dans le treillis : $(L, \sqsubseteq, \sqcup, \sqcap)$
  - $a \sqcup b$ existe et est la borne supérieure des deux éléments a et b.
  - $a \sqcap b$ existe et est la borne inférieure des deux éléments a et b.

Représentation par des diagrammes de Hasse





Pre/Post Points-fixes

Pour une fonction f définie sur un treillis $(L, \sqsubseteq, \bot, \sqcup, \top, \sqcap)$ et à valeurs dans $(L, \sqsubseteq, \bot, \sqcup, \top, \sqcap)$.

Définition

- on appelle pré-point-fixe de f, tout élément x de L satisfaisant la propriété $f(x) \sqsubseteq x$
- on appelle post-point-fixe de f, tout élément x de L satisfaisant $x \sqsubseteq f(x)$.

- $PostFIX(f) = \{x | x \in L \wedge x \sqsubseteq f(x)\}$: l'ensemble des post-points-fixes de f.
- $PreFIX(f) = \{x | x \in L \wedge f(x) \sqsubseteq x\}$: l'ensemble des pré-points-fixes de f.
- $FIX(f) = \{x | x \in L \wedge f(x) = x\}$: l'ensemble des points-fixes de f.

Théorème de Knaster-Tarski (I)

Soit f une fonction monotone croissante sur un treillis complet $(T, \bot, \top, \bigvee, \bigwedge)$. Alors il existe un plus petit point fixe et un plus grand point fixe pour f.

- μf désigne le plus petit point fixe de f.
- νf désigne le plus grand point fixe de f.
- μf vérifie les propriétés suivantes : $f(\mu f) = \mu f$ et $\forall x \in T. f(x) \sqsubseteq x \Rightarrow \mu f \sqsubseteq x$ (μf est la borne inférieure des prépoints fixes de f ou $\bigwedge(Pre(f))$).
- νf vérifie les propriétés suivantes : $f(\nu f) = \nu f$ et $\forall x \in T. x \sqsubseteq f(x) \Rightarrow x \sqsubseteq \nu f$ (νf est la borne supérieure des postpoints fixes de f ou $\bigvee(Post(f))$).

Preuve

Posons $y = \bigwedge\{x | f(x) \sqsubseteq x\}$ et montrons que y est un point fixe de f et que y est le plus petit point fixe de f.

1. $f(y) \sqsubseteq y$
   - Pour tout x de $\{x | f(x) \sqsubseteq x\}$, $y \sqsubseteq x$
   - $f(y) \sqsubseteq f(x)$ (par monotonie de f).
   - $f(x) \sqsubseteq x$ (par définition de x).
   - $f(y) \sqsubseteq x$ (par déduction).
   - $f(y) \sqsubseteq \bigwedge\{x | f(x) \sqsubseteq x\}$ (par définition de la borne inférieure, $f(y)$ est un minorant).
   - $f(y) \sqsubseteq y$
2. $y \sqsubseteq f(y)$
   - $f(y) \sqsubseteq y$ (par le cas 1)
   - $f(f(y)) \sqsubseteq f(y)$ (par monotonie de f)
   - $f(y) \in \{x | f(x) \sqsubseteq x\}$
   - $y \sqsubseteq f(y)$ (par définition de la borne inférieure)
3. Conclusion : $f(y) = y$ ou $y \in FIX(f)$.

Preuve

- $f(y) = y$ et z tel que $f(z) = z$
  - $f(z) = z$ (par hypothèse sur z)
  - $f(z) \sqsubseteq z$ (par affaiblissement de l'égalité)
  - $z \in \{x | f(x) \sqsubseteq x\}$ (par définition de cet ensemble)
  - $y \sqsubseteq z$ (par construction)
- y est le plus petit point fixe de f.

lfp(f) et gfp(f)

- $\mu f = lfp(f) = \bigwedge\{x | f(x) \sqsubseteq x\}$
- $\nu f = gfp(f) = \bigvee\{x | x \sqsubseteq f(x)\}$

- $lfp(f)$ signifie *least fixed-point*
- $gfp(f)$ signifie *greatest fixed-point*

Positionnement des éléments

$$\top$$
$$|$$
$$x \in \{x | f(x) \sqsubseteq x\}$$
$$|$$
$$\nu f$$
$$|$$
$$x \in \{x | x = f(x)\}$$
$$|$$
$$\mu f$$
$$|$$
$$x \in \{x | x \sqsubseteq f(x)\}$$
$$|$$
$$\bot$$

- $\top = \sqcup\{x | f(x) \sqsubseteq x\}$
- $gfp(f) = \nu f = \sqcup\{x | x \sqsubseteq f(x)\}$
- $lfp(f) = \mu f = \sqcap\{x | f(x) \sqsubseteq x\}$
- $\bot = \sqcap\{x | x \sqsubseteq f(x)\}$

Preuve algébrique selon Hoare

- $x \in pre(f)$ **si, et seulement si,** $f(x) \leq x$ (définition)

Preuve algébrique selon Hoare

- $x \in pre(f)$ **si, et seulement si,** $f(x) \leq x$ (définition)
- $\forall a \in pre(f).\mu f \leq a$ (application de la définition de la borne inférieure d'un ensemble)

Preuve algébrique selon Hoare

- $x \in pre(f)$ **si, et seulement si,** $f(x) \leq x$ (définition)
- $\forall a \in pre(f).\mu f \leq a$ (application de la définition de la borne inférieure d'un ensemble)
- $\forall a \in pre(f).f(\mu f) \leq f(a) \leq a$ (croissance de f et propriété de a)

Preuve algébrique selon Hoare

- $x \in pre(f)$ **si, et seulement si,** $f(x) \leq x$ (définition)
- $\forall a \in pre(f).\mu f \leq a$ (application de la définition de la borne inférieure d'un ensemble)
- $\forall a \in pre(f).f(\mu f) \leq f(a) \leq a$ (croissance de f et propriété de a)
- $f(\mu f) \leq \mu f$ (application pour $a = \mu f$))

Preuve algébrique selon Hoare

- $x \in pre(f)$ **si, et seulement si,** $f(x) \leq x$ (définition)
- $\forall a \in pre(f).\mu f \leq a$ (application de la définition de la borne inférieure d'un ensemble)
- $\forall a \in pre(f).f(\mu f) \leq f(a) \leq a$ (croissance de f et propriété de a)
- $f(\mu f) \leq \mu f$ (application pour $a = \mu f$))
- $f(f(\mu f)) \leq f(\mu f)$ (croissance de f)

Preuve algébrique selon Hoare

- $x \in pre(f)$ **si, et seulement si,** $f(x) \leq x$ (définition)
- $\forall a \in pre(f).\mu f \leq a$ (application de la définition de la borne inférieure d'un ensemble)
- $\forall a \in pre(f).f(\mu f) \leq f(a) \leq a$ (croissance de f et propriété de a)
- $f(\mu f) \leq \mu f$ (application pour $a = \mu f$))
- $f(f(\mu f)) \leq f(\mu f)$ (croissance de f)
- $f(\mu f) \in pre(f)$ (définition des pré-points-fixes)

Preuve algébrique selon Hoare

- $x \in pre(f)$ **si, et seulement si,** $f(x) \leq x$ (définition)
- $\forall a \in pre(f).\mu f \leq a$ (application de la définition de la borne inférieure d'un ensemble)
- $\forall a \in pre(f).f(\mu f) \leq f(a) \leq a$ (croissance de f et propriété de a)
- $f(\mu f) \leq \mu f$ (application pour $a = \mu f$))
- $f(f(\mu f)) \leq f(\mu f)$ (croissance de f)
- $f(\mu f) \in pre(f)$ (définition des pré-points-fixes)
- $\mu f \leq f(\mu f)$ (définition de μf)
- $\mu f = f(\mu f)$ (définition de μf et propriété précédente)

Version constructive du théorème de Knaster-Tarski

Soit f une fonction monotone croissante sur un treillis complet $(T, \bot, \top, \bigvee, \bigwedge)$. Alors

1. La structure formée des points fixes de f sur T, $(fp(f), \sqsubseteq)$ est un treillis complet non-vide.

   $(fp(f) = \{x \in T : f(x) = x\}$

2. $lfp(f) \stackrel{def}{=} \bigvee_\alpha f^\alpha$ est le plus petit point fixe de f où :

$$\begin{cases} & f^0 & \stackrel{def}{=} & \bot \\ \alpha \textit{ ordinal successeur} & f^{\alpha+1} & \stackrel{def}{=} & f(f^\alpha) \\ \alpha \textit{ ordinal limite} & f^\alpha & \stackrel{def}{=} & \bigvee_{\beta<\alpha} f^\beta \end{cases}$$

3. $gfp(f) \stackrel{def}{=} \bigwedge_\alpha f_\alpha$ est le plus grand point fixe de f où

$$\begin{cases} & f_0 & \stackrel{def}{=} & \top \\ \alpha \textit{ ordinal successeur} & f_{\alpha+1} & \stackrel{def}{=} & f(f_\alpha) \\ \alpha \textit{ ordinal limite} & f_\alpha & \stackrel{def}{=} & \bigwedge_{\beta<\alpha} f_\beta \end{cases}$$

Computing the least fixed-point over a finite lattice

INPUT $F \in T \longrightarrow T$
OUTPUT $result = \mu.F$
VARIABLES $x, y \in T, i \in \mathbb{N}$
$\ell_0 : \{x, y \in T\}$
$x := \bot;$
$y := \bot;$
$i := 0;$
$\ell_{11} : \{x, y \in T \wedge x = F^i \wedge y = \bigcup_{k=0;k=i} F^k \wedge i \leq Card(T) \wedge i = 0\};$
WHILE $i \leq Card(T)$
  $\ell_1 : \{x, y \in T \wedge x = F^i \wedge y = \bigcup_{k=0;k=i} F^k \wedge i \leq Card(T)\};$
  $x := F(x);$
  $\ell_2 : \{x, y \in T \wedge x = F^{i+1} \wedge y = \bigcup_{k=0;k=i} F^k \wedge i \leq Card(T)\};$
  $y := x \sqcup y;$
  $\ell_3 : \{x, y \in T \wedge x = F^{i+1} \wedge y = \bigcup_{k=0;k=i+1} F^k \wedge i \leq Card(T)\};$
  $i := i{+}1;$
  $\ell_4 : \{x, y \in T \wedge x = F^i \wedge y = \bigcup_{k=0;k=i} F^k \wedge i \leq Card(T){+}1\};$
OD;
$\ell_5 : \{x, y \in T \wedge x = F^i \wedge y = \bigcup_{k=0;k=i} F^k \wedge i = Card(T){+}1\};$
$result := y;$
$\ell_6 : \{x, y \in T \wedge x = F^i \wedge y = \bigcup_{k=0;k=i} F^k \wedge i = Card(T){+}1 \wedge result = y\};$

Verification in action

- Identify the safety property S to check.
- Run the algorithm for computing μF.
- Check that $\mu F \subseteq S$ or $\overline{S} \cap \mu F = \varnothing$.
- Check that $BUG \cap \mu F = \varnothing$, when BUG is a set of states that you identify as *bad states*.

Problem

- The general case is either infinite or large ... approximations of μF.
- Computing over abstract finite domain
- How to compute when it is not decidable ?
- Develop a framework for defining sound abstractions of software systems under analysis.

Expression du problème à résoudre

- S is the set of states
- $(\mathcal{P}(S \times S), \subseteq)$ is a complete lattice.
- $R \subseteq S \times S$ is a binary relation over S simulating the computation or the transition as $Next(x, x')$.
- $F \in \mathcal{P}(S \times S) \rightarrow \mathcal{P}(S \times S)$ is defined by the following expression :
  - $X \subseteq S \times S$
  - $I = \{(s, s) | s \in S\}$
  - $F(X) = I \cup R; X$

Transitive closure of R

- F is a monotonous function.
- F has a least-fixed point denoted μF.
- $\mu F = R^\star$
- $\forall n \in \mathbb{N} : F^{n+1}(\varnothing) = \bigcup_{i=0}^{n}$
- $\bigcup_{n \geq 0} F^n = R^\star$

Checking safety

- $G \subseteq S$ is the set of Good states.
- $I \subseteq S$ is the set of initial states.
- $\forall s_0, s \in S : s_0 \in I \wedge R^\star(s_0, s) \Rightarrow s \in G$ (Expression of safety of G)
- $\forall s \in S : (\exists s_0 . s_0 \in I \wedge R^\star(s_0, s)) \Rightarrow s \in G$
- $R^\star[I] \subset G$
- $R^\star[I] = \bigcup_{n \geq 0} F^n[I] = \bigcup_{n \geq 0} G^n$
- $\bigcup_{n \geq 0} G^n \subseteq G$ (Expression de la safety)

Techniques fo checking

$\bigcup_{n \geq 0} G^n \subseteq G$ is checked using at least two possible techniques :

- Bounded Model Checking
- k-Induction
- ...

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Exemple de contrats

(simple contrat)

Listing 18 – kindsession2/contract0.lus

```
node g () returns ();
(*@contract
  assume true;
  guarantee true;
*)
const n : int  = 7;
var t : int;
let
        t = n;

tel;
```

Exemple de contrats

(simple contrat)

Listing 19 – kindsession2/contract1.lus

```
contract countSpec(trigger: bool; val: int) returns (count: int ; error: bool) ;

let
  assume val >= 0;
  var initVal: int = val -> pre(initVal);
  var once: bool = trigger or (false -> pre once) ;
  guarantee count >= 0 ;

mode still_zero (
    require not once ;
    ensure count = initVal ;
  ) ;

mode gt (
    require not ::still_zero ;
    ensure count > 0 ;
  ) ;
tel
```

assumes

An assumption over a node n is a constraint one must respect in order to use n legally. It cannot depend on outputs of n in the current state, but referring to outputs under a pre is fine.
The idea is that it does not make sense to ask the caller to respect some constraints over the outputs of n, as the caller has no control over them other than the inputs it feeds n with.
The assumption may however depend on previous values of the outputs produced by n. Assumptions are given with the assume keyword, followed by any legal Boolean expression :

-

guarantees

Unlike assumptions, guarantees do not have any restrictions on the streams they can depend on. They typically mention the outputs in the current state since they express the behavior of the node they specified under the assumptions of this node.
Guarantees are given with the guarantee keyword, followed by any legal Boolean expression :

modes

A mode (R,E) is a set of requires R and a set of ensures E.
Modes are named to ease traceability and improve feedback.

(mode)

Listing 20 – kindsession2/mode.lus

```
mode <id> (
  [require <expr> ;]*
  [ensure  <expr> ;]*
);
```

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Summary

1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes
10. Contrats
11. Lustre in Kind2
    - Back to Lustre Programs
    - Basic definitions
    - Temporal Operators
    - Declarative semantics
    - Composite Types
12. Contracts for LUSTRE in Kind2
13. Propriétés des programmes LUSTRE

Semantical Concepts for Reactive Programming

4	4	4	...	4	...
x	x_0	x_1	...	x_n	...
y	y_0	y_1	...	y_n	...
x+y	x_0+y_0	x_1+y_1	...	x_n+y_n	...

x	x_0	x_1	...	x_n	...
pre x	NIL	x_0	...	x_{n-1}	...

x	x_0	x_1	...	x_n	...
y	y_0	y_1	...	y_n	...
x->y	x_0	y_1	...	y_n	...

- `nat = 0 -> (1 + pre nat)`

Programmes LUSTRE

- **Un programme LUSTRE s'appelle un nœud ou NODE.**
- **Un programme LUSTRE dénote une suite infinie de valeurs comme suit :** $(x_0\ x_1\ x_2\ \ldots)$
- **Deux opérateurs sur les programmes :**
  - $\mathbf{pre}$
  - $\longrightarrow$
- $\forall n \geq 0. CUP_{n+1} = CUP_n + 1$ **s'écrira** $CUP = 0 \longrightarrow (1 + \mathbf{pre}(CUP))$
- **et produira la suite** $(0\ 1\ 2\ \ldots)$**.**
- $FIB = 0 \longrightarrow 1 \longrightarrow (\mathbf{pre}(FIB) + \mathbf{pre}(\mathbf{pre}(FIB)))$

Programmes LUSTRE

- $$\begin{array}{l} node \quad EDGE(X : bool) \quad returns \quad (Y : bool) \\ let \\ \quad Y = false \rightarrow \; X \; and \; not \; pre(X) \\ tel \end{array}$$

  désigne la suite $(false, x_1 \wedge \neg x_0, x_2 \wedge \neg x_1, \; ...)$

- **Counter**

  $C = 0 \longrightarrow \mathbf{pre}(C)+1$ **renvoie la suite des naturels**

  $C = 0 \longrightarrow if \; X \; then \; \mathbf{pre}(C)+1 \; else \; \mathbf{pre}(C)$

  compte le nombre d'occurrences de X **qui sont vraies.**

  On ignore la valeur initiale

  $PC = 0 \longrightarrow \mathbf{pre}(C)$

  $C = if \; X \; then \; PC+1 \; else \; PC$

Summary

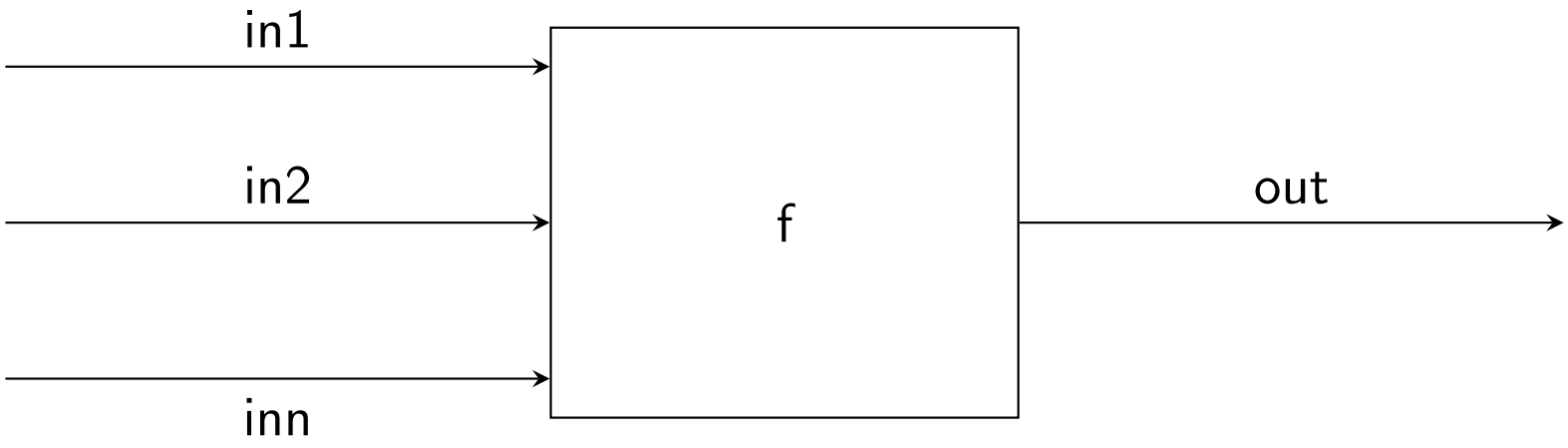
1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes
10. Contrats
11. Lustre in Kind2
    - Back to Lustre Programs
    - Basic definitions
    - Temporal Operators
    - Declarative semantics
    - Composite Types
12. Contracts for LUSTRE in Kind2
13. Propriétés des programmes LUSTRE

Lustre Nodes

- Lustre : a synchronous data-flow language for modeling and implementing reactive systems
- or a declarative parallel programming language
- or as an executable specification language.

Lustre Nodes

- Lustre : a synchronous data-flow language for modeling and implementing reactive systems
- or a declarative parallel programming language
- or as an executable specification language.
- The basic unit of computation in a Lustre program, or model, is a node, which is just a stream transformer : it takes streams of input and produces streams of output.



Forme générale d'un module LUSTRE

node $f(x_1 : \alpha_1, \ldots, x_n : \alpha_n)$ returns $(y_1 : \beta_1, ..., y_m : \beta_m)$
var $z_1 : \gamma_1, ..., z_k : \gamma_k;$
let
$z_1 = \delta_1; ...; z_k = \delta_k;$
$y_1 = \epsilon_1; ...; y_m = \epsilon_m;$
assert $P_1; \ldots;$ assert $P_l;$
tel

Reminding the synchronous model

- At any time, a node reads its input and generates its output step by step. in discrete cycles defined by an abstract global clock.
- At each cycle, all output values are assumed to be computed instantaneously from the current input and state values.
- By default, all nodes in a model compute synchronously and in parallel according to the global clock.
- A stream is an infinite sequence of values, all of the same (given) type.

Lustre node

A Lustre node can be viewed as modeling an infinite sequence of discrete timesteps, where at each timestep, each node variable takes its next value.

Example of the fusion of two streams(I)

Listing 21 – .lus

```
  node Fusion(x: int; y: int) returns (z: int);
  var u:int;
  v:int;
  let
z = x + y;
u=1;
v=2;
tel
```

The meaning or semantics is defined as follows :
$\forall n \in \mathbb{N}. z_n = x_n + y_n$

Example of the diffusion of one stream (II)

Listing 22 – .lus

```
node Diffusion ( x : int ) returns ( odd : int ; even :
 let
 even = x + x ;
 odd  = x + x +1 ;
 tel
```

The meaning or semantics is defined as follows :

$\forall n \in \mathbb{N}.even_n = x_n + x_n$

$\forall n \in \mathbb{N}.odd_n = x_n + x_n + 1$

Example of the diffusion of one stream (III)

Listing 23 – .lus

```
const C: int = 2;
node Combine ( x : int ; y : int ) returns ( z : int )
var l : int ;
let
l = C * y ;
z = x + l ;
tel
```

- Defining constants
- Defining local streams

Example of the diffusion of one stream (IV)

Listing 24 – .lus

```
const A : int = 2;
node E( x : int ; y : int ) returns ( z : int ) ;
var l : int ;
    let
     l = A * y ;
      z = x + l ;

 check y > 0  => l > y ; -- invariant
 check z > 0; -- not invariant
 tel
```

- Adding directives for checking properties on the current values
- Playing with SMT solvers and model-checkers
- Making analysis possible

Primitive Types

- primitive types are bool, int, and real
- the Boolean operators not, and, or, xor, and =¿ (implies), as well as the arithmetic operators +, - (both unary and binary), *, /, mod, and div (integer division), all with the expected arity and (pointwise) semantics.
- Lustre supports *if-then-else* expressions with the syntax $if < expr0 > then < expr1 > else < expr2 >$ where $< expr0 >$ has type bool and $< expr1 >$ and $< expr2 >$ must have the same type.

Summary

1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes
10. Contrats
11. Lustre in Kind2
    - Back to Lustre Programs
    - Basic definitions
    - Temporal Operators
    - Declarative semantics
    - Composite Types
12. Contracts for LUSTRE in Kind2
13. Propriétés des programmes LUSTRE

Stream computations for expression `1 -> (1 + pre x)`

	0	1	2	...	n
`1`	1	1	...	1	...
`x`	x_0	x_1	x_2	...	x_n
`pre x`	?	x_0	x_1	...	x_{n-1}
`1+pre x`	1+?	$1+x_0$	$1+x_1$	...	$1+x_{n-1}$
`1+(1+pre x)`	1	$1+x_0$	$1+x_1$	...	$1+x_{n-1}$

Stream computations for expression `1 -> (1 + pre x)`

Listing 25 – .lus

```
node E( x : int) returns ( y : int) ;

let
    y =  (1 + pre x);

 tel
```

Listing 26 – .lus

```
node E( x : int) returns ( y : int) ;

let
    y = 1 -> (1 + pre x);

 tel
```

Creating list of even numbers

Listing 27 – .lus

```
node even(_ : bool) returns (e: int);
var n:int;

let
  -- all the natural even numbers
  e = 0 -> (2 + pre e);
  n = 0 -> (1  + pre n);
  check 2*n = e;
tel
```

Checking the result by comparing two tsreams : $2{\cdot}n = even$.

Summary

1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes
10. Contrats
11. Lustre in Kind2
    - Back to Lustre Programs
    - Basic definitions
    - Temporal Operators
    - Declarative semantics
    - Composite Types
12. Contracts for LUSTRE in Kind2
13. Propriétés des programmes LUSTRE

Set of equations

Listing 28 – fact.lus

```
node Factorial(_:bool) returns (F: int);
var N: int;
 let
        -- all the factorial numbers
        F = 1 -> N * (pre F);
        -- all the natural numbers
        N = 0 -> (pre N) + 1;

 tel
```

- Lustre has a declarative semantics, meaning that the order of equations in node bodies does not matter.
- node equations should not be viewed imperatively as assignments.
- a node body is a set of stream constraints of the form $< var >=< expr >$

Circularity

Listing 29 – c8.lus

```
node Circular () returns ( out1 , out2 : int ) ;
let
 out1 = out2 + 1;
 out2 = out1 - 1;
 tel
```

- No values for steams count1 and count2.
- Cyclic dependency detected in definition of identifiers : out1, out2.

Summary

1. Introduction for reactve systems and cyber-physical systems
   - Evaluation des besoins
   - Le paradigme synchrone
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes
10. Contrats
11. Lustre in Kind2
    - Back to Lustre Programs
    - Basic definitions
    - Temporal Operators
    - Declarative semantics
    - Composite Types
12. Contracts for LUSTRE in Kind2
13. Propriétés des programmes LUSTRE

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Example of contrants

(simple contrat)

Listing 30 – kindsession2/contract0.lus

```
node g () returns ();
(*@contract
  assume true;
  guarantee true;
*)
const n : int  = 7;
var t : int;
let
        t = n;

tel;
```

assumes

An assumption over a node n is a constraint one must respect in order to use n legally. It cannot depend on outputs of n in the current state, but referring to outputs under a pre is fine.
The idea is that it does not make sense to ask the caller to respect some constraints over the outputs of n, as the caller has no control over them other than the inputs it feeds n with.
The assumption may however depend on previous values of the outputs produced by n. Assumptions are given with the assume keyword, followed by any legal Boolean expression :

-

guarantees

Unlike assumptions, guarantees do not have any restrictions on the streams they can depend on. They typically mention the outputs in the current state since they express the behavior of the node they specified under the assumptions of this node.
Guarantees are given with the guarantee keyword, followed by any legal Boolean expression :

modes

A mode (R,E) is a set of requires R and a set of ensures E.
Modes are named to ease traceability and improve feedback.

(mode)

Listing 31 – kindsession2/mode.lus

```
mode <id> (
  [require <expr> ;]*
  [ensure  <expr> ;]*
);
```

division

(division)

Listing 32 – division.lus

```
node safe_div(x: int; y: int) returns (z: int);
(*@contract
  assume y <> 0;
  guarantee z * y = x;
*)
let
  z = x / y;
teloggle.lus
```

toggle

(toggle)

Listing 33 – toggle.lus

```
node toggle() returns (o: bool);
(*@contract
  guarantee o <> pre(o);
*)
let
  o = false -> not pre(o);
tel
```

bounded counter

(boundedcounter)

Listing 34 – boundedcounter.lus

```
const a: int = 4;
node bounded_counter(_:bool) returns (c: int);
(*@contract
  guarantee c >= 0 and c < a;
*)

let
  c = 0 -> (pre(c) + 1) mod 5;
tel
```

alarm

(alarm)

Listing 35 – alarm.lus

```
node system(x: int) returns (alarm: bool);
(*@contract
  guarantee not alarm or x > 100;
*)
let
  alarm = x > 100;
tel
```

hypo

(hypo)

Listing 36 – hypo.lus

```
node example(x: int) returns (y: int);
(*@contract
  assume x >= 0;
  guarantee y >= 0;
*)
let
  y = x; -- simple propagation
tel
```

request

(request)

Listing 37 – request.lus

```
node request_response(req: bool) returns (resp: bool);
(*@contract
  guarantee req => (true -> pre(resp));
*)
let
  resp = false -> pre(req);
tel
```

Exemple de contrats

(simple contrat)

Listing 38 – kindsession2/contract1.lus

```
contract countSpec(trigger: bool; val: int) returns (count: int ; error: bool) ;

let
  assume val >= 0;
  var initVal: int = val -> pre(initVal);
  var once: bool = trigger or (false -> pre once) ;
  guarantee count >= 0 ;

mode still_zero (
    require not once ;
    ensure count = initVal ;
  ) ;

mode gt (
    require not ::still_zero ;
    ensure count > 0 ;
  ) ;
tel
```

Summary

1. Introduction for reactve systems and cyber-physical systems
2. Modélisation synchrone (I)
3. Le langage LUSTRE
4. Tools for Synchronous Modelling
5. Exemples de programmes LUSTRE
6. Modélisation synchrone (II)
7. Vérification de programmes LUSTRE
8. Propriétés des programmes LUSTRE
9. Sommaire sur les points-fixes

Propriétés des programmes LUSTRE

()

Listing 39 – kindsession2/power2.lus

```
-- power2(_) contains all the square  numbers in increasing order
-- where 0^2 = 0
--       (n+1)^2 = n^2 +2*n + 1
node power2() returns (P: int);
var W: int;
let
  -- all the natural even numbers
  W = 1 -> (pre W) + 2;
  P = 0 -> (pre P) + (pre W);
tel
```

Propriétés des programmes LUSTRE

()

Listing 41 – kindsession2/power2.lus

```
-- power2(_) contains all the square  numbers in increasing order
-- where 0^2 = 0
--       (n+1)^2 = n^2 +2*n + 1
node power2() returns (P: int);
var W: int;
let
  -- all the natural even numbers
  W = 1 -> (pre W) + 2;
  P = 0 -> (pre P) + (pre W);
tel
```

(`--enable BMC --enable IND`)

Listing 42 – kindsession2/power2prop.lus

```
-- power2(_) contains all the square  numbers in increasing order
-- where 0^2 = 0
--       (n+1)^2 = n^2 +2*n + 1
include "power2.lus"
node power2prop() returns (P: int);
var P2,PP2,N:int;
var PROP:bool;
let
  PP2 = power2();
  N = 0 -> (pre N)+1;
  P2 = N*N;
  PROP  = P2 = PP2;
  check PROP;
tel
```

()

Listing 43 – kindsession2/greycounter.lus

```
node greycounter (reset: bool) returns (out: bool);
var a, b: bool;
let
  a = false -> (not reset and not pre b);
  b = false -> (not reset and pre a);
  out = a and b;

tel
```

()

Listing 44 – kindsession2/intcounter.lus

```
node intcounter (reset: bool; const max: int) returns (out: bool);
var t: int;
let
  t = 0 -> if reset or pre t = max then 0 else pre t + 1;
  out = t = 2;
tel
```

Propriétés des programmes LUSTRE

(Valid property)

Listing 45 – kindsession2/ex1.lus

```
/* include   */
include "greycounter.lus"

/* include   */
include "intcounter.lus"


node top (reset: bool) returns (OK, OK2: bool);
var b, d: bool;
let
  b = greycounter(reset);
  d = intcounter(reset, 3);
  OK = b = d;
  --%PROPERTY OK;
tel
```

Propriétés des programmes LUSTRE

(Invalid property)

Listing 46 – kindsession2/ex2.lus

```
/* include   */
include "greycounter.lus"

/* include   */
include "intcounter.lus"


node top (reset: bool) returns (OK, OK2: bool);
var b, d: bool;
let
  b = greycounter(reset);
  d = intcounter(reset, 3);
  OK = b = d;
  OK2 = not d;
  --  %PROPERTY OK;
  --%PROPERTY OK2;
tel
```