Langages synchrones

Jérôme Hugues (ex. ENST)
Emmanuelle Encrenaz-Tiphène

(emmanuelle.encrenaz@lip6.fr)

sept 2018

Master d'Informatique
Spécialité SAR
SPECIF – Langages Synchrones

4. Vérification statique et Compilation

a) Vérifications statiques : Analyse de causalité

b) Compilation en code exécutable

1. Retour sur les systèmes séquentiels
2. Programme « boucle infinie »
3. Programme « Automate »
4. Pb de modularité de la compilation

Master d'Informatique
Spécialité SAR
SPECIF – Langages Synchrones

a) Respect de la causalité

Master d'Informatique
Spécialité SAR
SPECIF – Langages Synchrones

Analyse statique de programmes

- **Objectifs**
  - Cohérence des types de données, initialisation des variables, etc
  - Cohérence des horloges en Lustre
  - Respect de la causalité
    - *Erreur si la valeur d'un flot dépend d'elle-même à l'instant courant*
- **Propriétés fondamentales des programmes synchrones**
  - Réactivité : pour toute entrée, au moins un comportement possible
  - Déterminisme : pour toute entrée, au plus un comportement possible
- **Idéalement : avoir réactivité et déterminisme**
  - Assure progression du système (pas d'état bloqué)
  - Assure reproductibilité, fiabilité, etc.

Exemples de code erroné

- **Non réactivité d'un programme**
  - Lustre: **X = not X**
  - Pas de solution => doit être rejeté
  - Un tel système ne peut évoluer
- **Non déterminisme**
  - Lustre: **X = X**
  - Plusieurs solutions => doit être rejeté

Construction du graphe de dépendances de données

- **Graphe de dépendance G = <E,VC,VS>**
  - E : ensemble des variables du nœud, avec leur estampillage temporel
  - VC ⊆ E x E : (a,b) ∈ V, noté a → b si b est résultat d'un opérateur combinatoire appelé avec a ou si a est renommé en b (par instanciation)
  - VS ⊆ E x E : (a,b) ∈ V, noté a ⇒ b si b est résultat d'un opérateur séquentiel appelé avec b
- **Exemples : construire les graphes de dépendances associés aux nœuds suivants**

```
node N1 (a,b : bool) returns (s : bool);
var pb : bool;
let
    s = a and pb;
    pb = pre b;
tel
```

```
node N2 (x,y : bool) returns (t : bool);
var px : bool;
let
    t = px or y;
    px = pre x;
tel
```

```
node main(p,q : bool) returns (v : bool);
var r : bool;
let
    v = N1(r,q);
    r = N2(v,p);
tel
```

Graphe de dépendances

Une Solution Simple

- **Solution :**
  - interdire les boucles d'opérateurs ne comprenant pas au moins un décalage temporel
  - Solution retenue par Lustre
- **Avantage :**
  - critère simple à vérifier par analyse du graphe de dépendances des données.
- **Inconvénient: trop discriminant**
  - Rejette des programmes réactifs et déterministes

```
X = if C then Y else A;
Y = if C then A else X;
```

- **Attention : l'instanciation de deux nœuds sans boucle de causalité peut introduire une boucle de causalité !**

b) Génération de code séquentiel

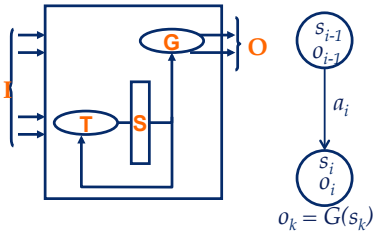
1. Retour sur les systèmes séquentiels
2. *(Problème de la compilation séparée)*
3. Code en boucle
4. Code automate

Master d'Informatique
Spécialité SAR
SPECIF – Langages Synchrones

Machine de Moore – Machine de Mealy

Machine de Moore M = < S, I, O, T, G, S_0>

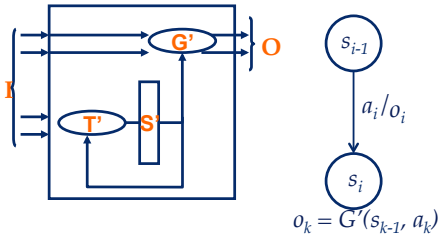
- *S = ensemble fini d'états (états de contrôle)*
- *I = ensemble fini des signaux d'entrée*
- *O = ensemble fini des signaux de sortie*
- *T = fonction de transition : $S \times 2^I \rightarrow S$*
- *G = fonction de génération : $S \rightarrow 2^O$*
- *S_0 = état initial*



Séquence d'entrée : $a_1, a_2, \ldots a_n, \ldots$
Exécution : $(s_0,a_1,s_1),(s_1,a_2,s_2) \ldots$
Séquence de sortie : $G(s_0), G(s_1), \ldots G(s_n), \ldots$

Machine de Mealy M' = < S', I, O, T', G', S'_0 >

- *S' = ensemble fini d'états (états de contrôle)*
- *I = ensemble fini des signaux d'entrée*
- *O = ensemble fini des signaux de sortie*
- *T' = fonction de transition : $S' \times 2^I \rightarrow S'$*
- *G = fonction de génération : $S \times 2^I \rightarrow 2^O$*
- *S_0 = état initial*



Séquence d'entrée : $a_1, a_2, \ldots a_n, \ldots$
Exécution : $(s'_0,a_1,s'_1),(s'_1,a_2,s'_2) \ldots$
Séquence de sortie : $G'(s_0,i_1), G'(s_1,i_2), \ldots G'(s_n,i_{n+1}),.$

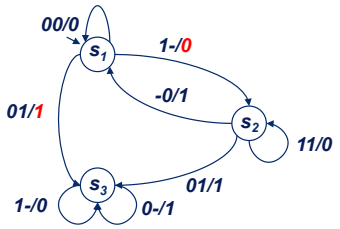
Equivalence des machines

- **Deux machines sont dites équivalentes si pour chaque séquence d'entrée, elles produisent la même séquence de sortie.**
- **Du point de vue des séquences d'entrées/sorties produites, et sans considération des connexions de la machine avec d'autres machines (ou alors respectant l'hypothèse synchrone) :**
  - Pour toute Machine de Moore M, il existe une machine de Mealy M' équivalente
    - *Garder les mêmes états et transitions que M, ajouter o_{i+1} sur toutes les transitions entrantes dans l'état s_{i+1}*
  - Pour toute Machine de Mealy M', il existe une Machine de Moore M équivalente
    - *Répliquer chaque état de M' en 2^O états de M, chacun associé à une configuration particulière de 2^O.*
    - *Pour chaque transition (s,a/o,s') de M', construire ((s,-),a,(s',o))*
    - *Tous les états (s_0,-) sont initiaux*

Mealy – Moore

Une machine de Mealy M' = < S', I, O, T', G', S'_0>

- **S' = {s'_1, s'_2, s'_3}, S_0 = {s_1}**
- **I = {i_1,i_2}, O = {o}**
- **T' = <s'_1,00> → s'_1, <s'_1,01> → s'_3, <s'_1,1-> → s'_2**
  <s'_2,-0> → s'_1, <s'_2,11> → s'_2, <s'_2,01> → s'_3
  <s'_3,--> → s'_3
- **G' = <s'_1,00> → 0, <s'_1,01> → 1, <s'_1, 1-> → 0**
  <s'_2,-0> → 1, <s'_2,11> → 0, <s'_2,01> → 1
  <s'_3,1-> → 0, <s'_3,0-> → 1



Exercices

- **Construire une Machine de Mealy reconnaissant les mots binaires composés de suites paires de 0 ou de 1 consécutifs. (La machine produit 1 tant qu'elle n'a pas rencontré de chaine maximale impaire de 1 ou de 0).**
- **Construire une Machine de Moore équivalente.**

Machine de Mealy interprétée

- **La machine interagit avec un espace de données D lors du franchissement de transitions.**
  - **D** : un ensemble de variables typées; v une valuation sur D (une fonction associan à chaque variable une valeur de son type), Val l'ensemble des valuations sur D
  - **Garde** (expression booléenne *g*) : transition franchissable si l'évaluation de *g* pour les valeurs des variables de l'espace de donnée renvoie VRAI.
  - **Action** (affectations *a*) : franchissement de la transition produit un calcul dont le résultat est stocké dans les variables de D, par affectation.
  - Une commande gardée : un couple (g,a) : (Val -> B) x (Val -> Val)
  - **GA** l'ensemble des commandes gardées sur D
  - Machine de Mealy interprétée sur D : **< I,O,L,K, l_0>**
    - *I,O,L : ensemble fini des variables d'entrées / sorties, de points de contrôle (locations)*
    - *K: ensemble fini de transitions (L x 2^I x GA x 2^O x L)*
    - *l_0 : location initiale*

Machine de Mealy interprétée

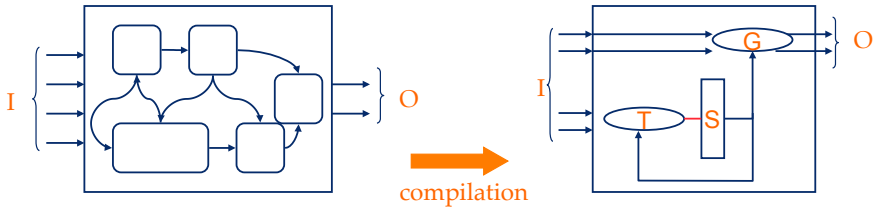
- **Exemple : Machine de Mealy interprétée reconnaissant les mots binaires composés de chaines maximales paires de 0 ou de 1 consécutifs.**
- **i = nouvelle lettre du mot binaire**
- **o = sortie (1 tant que le mot lu ne contient pas de chaine complète impaire de 0 ou 1 consécutifs)**
- **D = {lettre_prec} lettre lue sur la réaction précédente**

Automate associé

- **On peut associer une machine de Mealy (sans espace de données) à une machine interprétée**
- **MI = <I,O,L,K,l_0>, on construit MM = <S,I,O,T,G,s_0>**
  - S ⊆ L x Val
  - T ⊆ S x 2^I x S tq : si (l,i,(g,a),o,l') ∈ K,
    alors ((l,v),i,(l',v')) ∈ T avec g(v) = TRUE, a(v) = v'
  - s_0 = (l_0,v_0)
- **Dépliage : incorporation de variables dans les locations**
- **Le dépliage peut être partiel (seules certaines variables de D sont incorporées dans les états de la Machine dépliée)**
- **Le repliage complet produit un automate interprété à une seule location (code en boucle)**

Compilation en code C exécutable

La description Lustre est transcrite en un programme exécutable (C)



Description réactive parallèle

Description séquentielle (machine de Mealy interprétée)

Difficultés :

1- Trouver un ordonnancement global des calculs représentant les définitions de flots
2- Identifier les variables du programme séquentiel
3- Compromis temps d'exécution des réactions, place du code en mémoire

Structure du programme séquentiel

- **Code en boucle**

```
var I, O, S;
proc P_step() … ;      // traitement :
                       // évaluation de toutes les équations
                       // modification de S

S := S0;          // initialisation

foreach step do // boucle infinie
    read(I);
    P_step();
    write(O);
end foreach
```

Génération de code en boucle

- **Expansion des noeuds du programme**
  - Applatissement de la hiérarchie : 1 système d'équations concurrentes
- **Identification des variables du code produit**
  - Entrées, Sorties,
  - *pre (exp) , ->*
    - points de coupure du graphe de dépendance des variables
    - Éléments mémorisants
- **Ordonnancement des équations du système**
  - code sans boucle (coupées par pre)

  → Résolution des équations fournit un ordonnancement possible
  - Substitution des variables x = f(y); y = g(z); remplacé par x = f (g (z))
- **Affectations**
  - Les définitions « = » deviennent des affectations « := »
    - *x = pre (0 -> x) + 1; devient x = 0; x = x + 1;*
  - Les alternatives if then else sont traduites par l'opérateur triadique ? :

Exemple


```
node compter (tick, top: bool) returns (cpt:int);
var i : int;
let
  cpt = 0 -> if pre top then i
             else if tick then pre cpt + 1 else pre cpt;
  i = if tick then 1 else 0;
tel;
```

- **Après identification de la mémoire, devient**

```
cpt = if init then 0
          else if ptop then i
          else if tick then pcpt + 1
          else pcpt;
i = if tick then 1 else 0
ptop = pre top;
pcpt = pre cpt;
init = true -> false;
```

Génération du code C


```
/* initialisations */
init = true;
while (1) {
/* lecture des entrées */
/* affectation de tick et top */

i = tick ? 1 : 0;
cpt = init ? 0
          : ptop ? i
          : tick ? pcpt + 1
          : pcpt;
/* modification de l'état */
ptop = top;
init = false;
pcpt = cpt;

/* écriture des sorties */
/* affichage de cpt */
}
```

BOUCLE EVALUATION

Optimisation des tests

P_STEP

pcpt est inutile

Exercice


```
node n1 (a,b : bool) return (s1,s2: bool);
let
  s1 = 0 -> a and pre b;
  s2 = 1 -> not pre s2;
tel

Construire la machine de Mealy interprétée (code en boucle)
  associée à n1.
```

Construction de l'automate implicite (code en boucle généré par Lustre)

- **Identification des éléments mémorisant du programme**
- **Réécriture en explicitant ces éléments mémorisants + ordonnancement des équations + état initial**

Transcription en C

- **a et b sont des variables d'entrée**
- **pb, ps2 et init sont des variables internes**
- **s1 et s2 sont des variables de sortie**

```
/* initialisation */


/* lecture des valeurs courantes de a et b */

/* réaction */




 /* écriture des valeurs courantes de s1 et s2 */
```

Analyse

- **Code en boucle**
  - Pas de structure : Évaluation systématique de l'intégralité du code à chaque itération

  → code lent
  - Machine de Mealy interprétée : toutes les variables entrées, internes et de sorties sont dans l'espace de données externe, les configuration d'entrées et de sortie sont recopiées sur les parties garde/sortie des transitions
- **Améliorations**
  - Réduction de la mémoire
    - *x et pre x sont partagées si toutes les équations lisant pre x peuvent être schedulées avant x …*
      - x = pre x + 1 → x := x + 1
    - *Définitions équivalentes :*
      - x = 1 -> pre x +1 ; y = 1 -> pre y +1 → x = 1 -> pre x + 1; y = x;
  - Amélioration de la structure du code
    - *Détection de portion de code exécutables uniquement à l'initialisation ou sur conditions bien identifiées (horloge particulière)*

Code en automate pour Lustre

- But : Produire un code qui n'évalue à chaque instant que la partie pertinente pour chaque variable : code exécuté dépend de *l'état courant* du système : compilation en automate

- Etat de l'automate
  Choix des variables (généralement celles mémorisées: sous la portée d'un pre)
  Dépliage du code selon les valeurs des variables

Exemple : exemple du nœud `compter` avec dépliage selon `init` et `ptop`

État initial S1 = [true/init] // ptop est à nil
i = tick ? 1 : 0; cpt = 0; devient cpt = 0;
Choix du prochain état: if (top) state = S2; else state = S3;

État S2 = [false/init, true/ptop]
i = tick ? 1 : 0; cpt = i; if (top) state = S2; else state = S3;

État S3 = [false/init, false/ptop]
i = tick ? 1 : 0; cpt = tick ? cpt + 1 : cpt; if (top) state = S2; else state = S3;

Nœud compter déplié

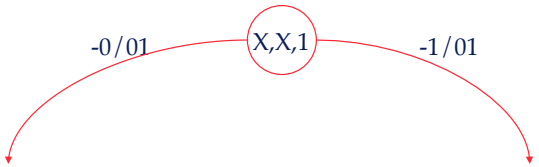

```
read (tick, top);

switch (state) {

case S1: cpt = 0;
                if (top) state = S2; else state = S2; break;

case S2: i = tick ? 1 : 0;
                 cpt = i;
                 if (top) state = S2; else state = S3; break;

case S3: i = tick ? 1 : 0;
                  cpt = tick ? cpt + 1 : cpt;
                  if (top) state = S2; else state = S3;
                  break;
}

write (cpt);
```

Retour sur le nœud n1


```
node n1 (a,b : bool) returns  (s1,s2: bool);
let
      s1 = 0 -> a and pre b;
      s2 = 1 -> not pre s2;
tel
```

```
/* réaction */
            s1 = if (init) 0 else (a && pb);
            s2 = if (init) 1 else  (! ps2);
            pb = b;
            ps2 = s2;
            init = 0;
```


- Dépliage de l'automate du nœud n1 selon pb, ps2, init



Analyse

- **Le code généré est de meilleur qualité, mais plus gros**
  - Code plat = l'automate représente le produit des composants synchrones qu'il contient
  - Les états de l'automate sont représentés explicitement – énumérés –
  - L'automate est en général non minimal
- **Les assertions sont prises en compte lors de la compilation pour supprimer certains états non accessibles**
  - (clauses `assert`)
- **Problèmes**
  - Choix des variables de dépliage : pre
  - Représentation efficace des fonctions booléennes
  - Explosion combinatoire
- **Solutions (cf. cours verif)**
  - Représenter le programme par des BDD (binary decision diagrams)
  - Minimisation d'automate
  - Compilation dirigée par la demande

Problème de la compilation séparée

- **Compilation séparée :**
  - Compiler indépendamment chaque nœud (ou module)
  - Les composer (concurrence / instanciation)
- **Réactivité : simultanéité de la source et de l'effet**
  - Ordonner les évaluations des gi,ti pour représenter g,t (programme causal)
- **Séquentialisation du parallélisme**
  - Le choix d'une séquentialisation *particulière* peut être incompatible avec le contexte dans lequel le nœud est appelé (une autre séquentialisation aurait pu convenir …)

Problème de la compilation séparée (2)

- **Exemple [Gonthier 85]**

```
node two_copies(a,b : int) returns (x,y : int);
let x = a; y = b tel

Node loop(t : int) returns (z : int);
var y : int;
let (y,z) = two_copies(t,y) tel
```

Problème de la compilation séparée (3)

- **Deux séquentialisations possibles pour two_copies :**

  Les deux définitions sont indépendantes; elles peuvent être exécutées dans n'importe quel ordre :

  1. x := a; y := b;
  2. y := b; x := a;

- **L'appel `(y,z) = two_copies(t,y)` induit un ordre d'évaluation des deux définitions. Ceci implique que seule la séquentialisation 1. est correcte pour ce contexte d'appel.**
- **Un autre appel (par ex. `(z,y) = two_nodes(y,t)`) peut nécessiter l'évaluation selon la séquentialisation 2.**

→ **On ne peut pas *a priori* générer une seule séquentialisation pour un nœud sans hypothèses sur le contexte d'appel**

→ **Dans *Scade*, la compilation séparée est possible mais les signaux connectant les modules traversent nécessairement au moins un opérateur de délai.**

d) Génération de circuit

Lustre et Esterel

Master d'Informatique
Spécialité SAR
SPECIF – Langages Synchrones

Deux types de circuits cible

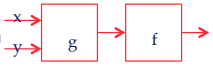
- **ASIC (Application Specif Integrated Circuit) numériques : circuit ad-hoc**
  - **Parties combinatoires** : réseaux de portes logiques (et, ou, non) sans boucles interconnectés par des signaux (équipotentielles)
  - **Parties séquentielles** : registres cadencés par une horloge globale, échantillonnant (sur condition d'écriture) la valeur de certains signaux
  - *Circuit rapide, faible surface, faible consommation*
  - *Coût de conception élevé, pas de reprise possible*
    - outils de synthèse pour blocs combinatoires / FSM (Moore,Mealy), placement, routage, estimations
    - Coût de la fabrication du circuit devient *prohibitif*
- **FPGA Field Programmable Gate Array) : circuit configurable par programmation**
  - **Parties combinatoires** : blocs fonctionnels programmables à n entrées et p sorties.
  - **Parties séquentielles** : bancs de registres
  - **Connexions** : réseau configurable entre les entrées/sorties des blocs fonctionnels et des bancs de registres
  - **Programmation**
    - Découper les fonctions à implanter en composition de fonctions implantables sur les blocs fonctionnels
    - Mémoriser la table de vérité de chaque bloc fonctionnel
    - Configurer le réseau d'interconnexion
  - *Circuit assez lent, grande taille, consommation élevée*
  - *Adapté au prototypage d'applications, reconfigurations possibles*

Compilation de Lustre

La description Lustre expansé / code en boucle est naturellement une description flot de données adaptée pour ASIC :

- Les types `int`, produit, tableaux sont convertis en vecteurs de bits
- Les opérations associées sont transposées sur vecteurs de bits

- Interface du nœud → interface du composant
- Expression dans pre et valeurs initiales (mémoire) → registre
- Expression de chaque sortie → un réseau d'opérations logiques (dépendant des entrées primaires et des registres) :
  - pour chaque opérateur Lustre,
    - associer le circuit opérateur correspondant.
    - opérandes de l'opérateur → entrées
    - Expression → sortie
    - Composition de fonction → séquentialisation (f(g(x,y)) transcrit en



- Partage de calcul est le même que celui exprimé dans la description Lustre

Compilation en circuit

- **Proposez une implantation en circuit du nœud Lustre n1.**