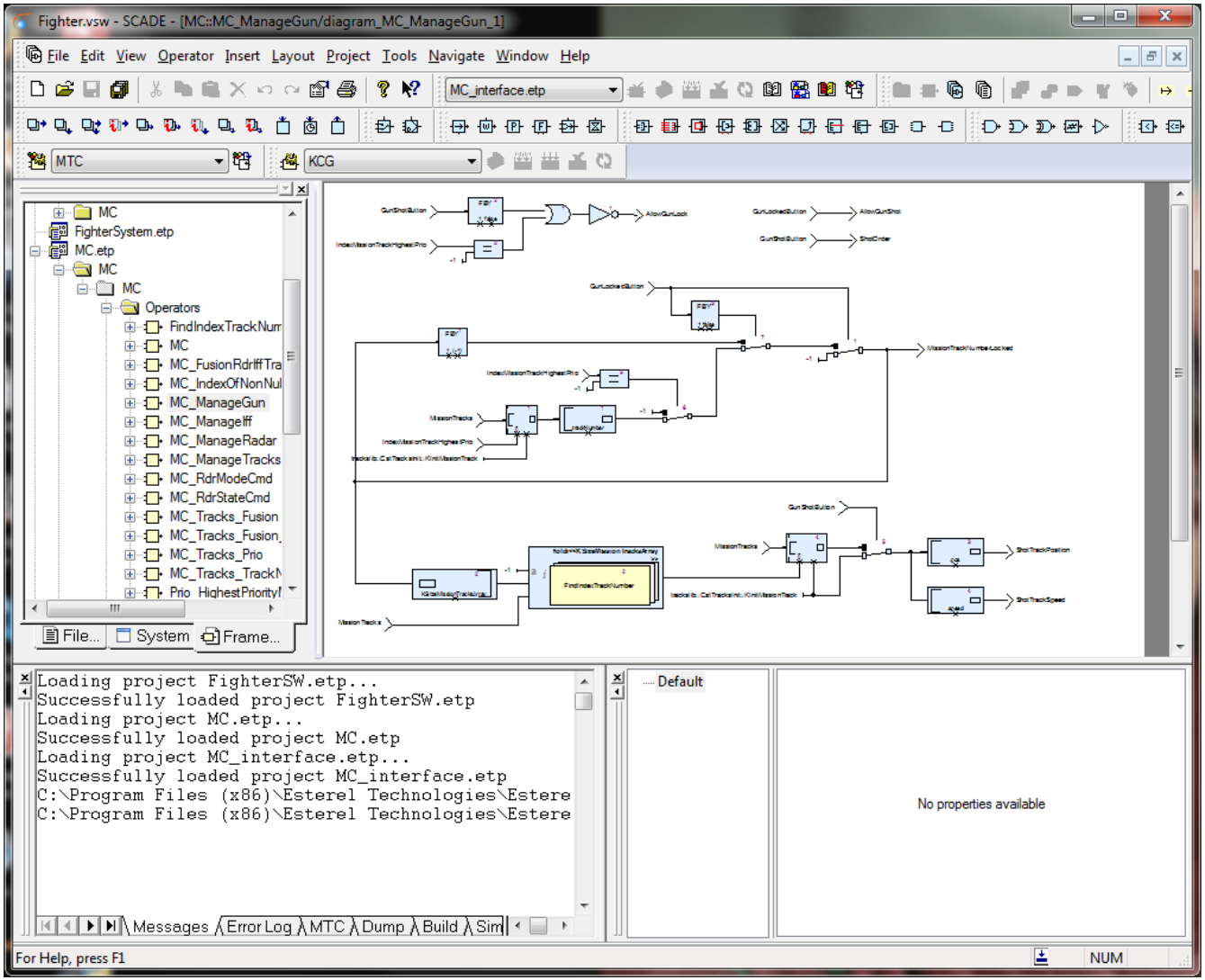
Le Langage Scade 6 pour les systèmes embarqués

De la conception à la compilation

Bruno Pagano
Esterel Technologies - ANSYS

Safety Critical Application Development Environment



Quelques contraintes de l'Embarqué Critique

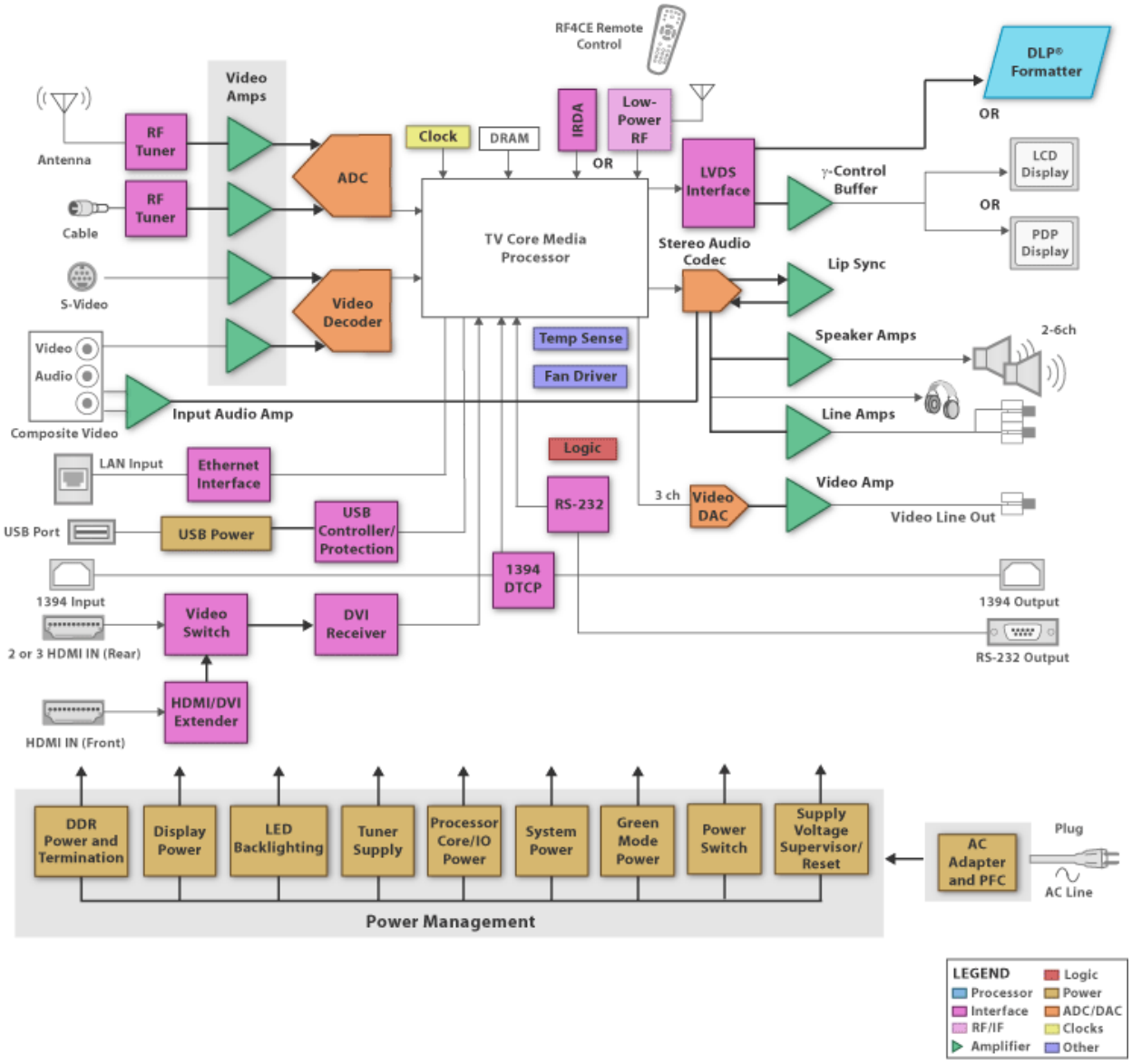
- Système réactif :
  - Destiné aux OS temps réels
  - Mémoire et WCET bornés
    - pas d'allocation dynamique
    - pas de récursion

- On attend du langage de programmation :
  - Description exhaustive
  - Bonnes fondations
  - Déterminisme
  - Relative simplicité
  - Traçabilité

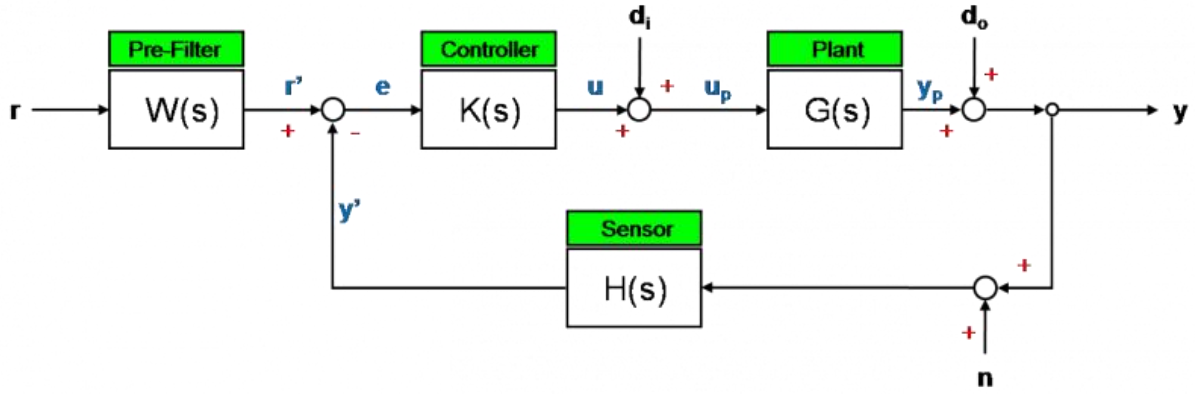
Les langages traditionnels

- C et Ada :
  - Langages généralistes et impératifs
  - Comportements non déterminés :
    - règles de codages (MISRA, etc.)
    - vérifications
- Scade :
  - Dédié aux systèmes réactifs
  - Orienté flot de données
  - Très contraint mais aucun comportement non-déterminé

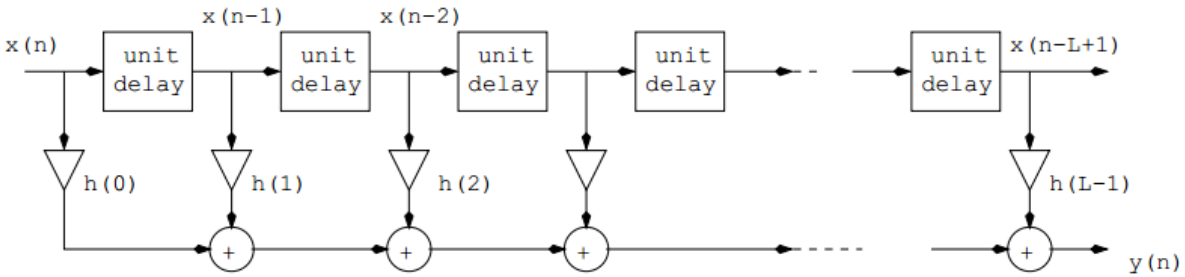
La tradition Schémas-Blocs (Block Diagram)



La tradition Schémas-Blocs (Block Diagram)



La tradition Schémas-Blocs (Block Diagram)



- SAO : Airbus
- Simulink® : Simulation
- SCADE Langage Synchrone
  Lustre
  Scade 6

Langage flot de données

- A chaque instant, après un instant initial :

  Produit un flot (flux) de sorties en fonction d'un flot d'entrées

$$f : IN^{\mathbb{N}} \rightarrow OUT^{\mathbb{N}}$$
$$f(i_0, i_1, \ldots, i_n, \ldots) = (o_0, o_1, \ldots, o_n, \ldots)$$

- Fonction de cycle (réaction)

$$f : IN \times STATE \rightarrow OUT \times STATE$$
$$f(\overrightarrow{in_p}, state_p) = (\overrightarrow{out_p}, state_{p+1})$$

code généré == fonction de cycle

Un exemple très basique

```
node F (a,b: int) returns (x,y: int)
var
   p : int;
let
  x = a + b ;
  y = if  p<0  then  –p  else  p ;
  p = a – b ;
tel
```

- des équations
- principe de substitution
- pas d'ordre
- ici, pas d'état

Un exemple très basique

```
node F (a,b: int) returns (x,y: int)
var
   p : int;
let
  x = a + b ;
  y = if  p<0  then  –p  else  p ;
  p = a – b ;
tel
```

t	t_0	t_1	…	t_n	…			
a	2	2	…	-2	…			
b	1	3	…	-3	…			
x	3	5	…	-5	…			
p	1	-1	…	1	…			
p<0	F	T	…	F	…			
y	1	1	…	1	…			

- des équations
- principe de substitution
- pas d'ordre
- ici, pas d'état

Un exemple très basique

```
node F (a,b: int) returns (x,y: int)
var
   p : int;
let
  x = a + b ;
  y = if  p<0  then  –p  else  p ;
  p = a – b ;
tel
```

- des équations
- principe de substitution
- pas d'ordre
- ici, pas d'état

```
/* ==  input structure  == */
typedef struct { int a; int b; } inC_F;

/* ==  context type  == */
typedef struct { int x;  int y; } outC_F;

/* F */
void F(inC_F *inC, outC_F *outC) {
  int p;

  outC->x = inC->a + inC->b;
  p = inC->a - inC->b;
  if (p < 0) {
    outC->y = - p;
  }
  else {
    outC->y = p;
 }
}
```

PRE : décalage temporel

y = **pre** x ;

t	t_0	t_1	t_2	t_3	t_4	…		
x	1	2	3	4	5	…		
pre x	nil	1	2	3	4	…		

z = 0 **->** y ;

z	0	1	2	3	4	…		

- Fonction de cycle : STATE

$$f : IN \times STATE \rightarrow OUT \times STATE$$
$$f(X_p, state_p) = (Y_p, Z_p, state_{p+1})$$

PRE : décalage temporel

y = **pre** x ;

z = 0 **->** y ;

t	t_0	t_1	t_2	t_3	t_4	…		
x	1	2	3	4	5	…		
pre x	nil	1	2	3	4	…		
z	0	1	2	3	4	…		

- Fonction de cycle : STATE = Bool x Int

$$state_O = (init_0 , memX_0) = (true, nil)$$

$$Y_p = memX_p$$

$$Z_p = \begin{cases} 0 & si\ init_p \\ Y_p & sinon \end{cases}$$

$$init_{p+1} = false$$

$$memX_{p+1} = X_p$$

PRE : décalage temporel

```
/* ==  no input structure  == */


/* =====  context type  ========= */
typedef struct {
  /* -------  outputs  ---------------------- */
  int Y;
  int Z;
  /* ------- initialization variables  ------ */
 bool init;
 /* ------- local memories  -------------- */
  int rem_X;

} outC_Pre;
```

```
void Pre_reset(outC_Pre *outC)
{
 outC->init = kcg_true;
}


void Pre(int X, outC_Pre *outC)
{
 outC->Y = outC->rem_X;
 if (outC->init) {
  outC->init = kcg_false;
  outC->Z = 0;
 }
 else {
  outC->Z = outC->Y;
 }
 outC->rem_X = X;
}
```

Exemple : entiers naturels

```
node POS () returns (x: int)
  x = 1 -> 1 + pre x ;

node NAT () returns (x: int)
  x = 0 -> POS() ;
```

- Un nœud Scade peut ne pas avoir d'entrées
- Principe de substitution
- Expansion / Non-expansion

WHEN: sous-échantillonage d'un flot

```
X = NAT() ;
C = (X mod 3 = 0) ;
Y = (2 * X) when C ;
```

t	t_0	t_1	t_2	t_3	t_4	t_5	t_6	…
X	0	1	2	3	4	5	6	…
2 * X	0	2	4	6	8	10	12	…
C	T	F	F	T	F	F	T	…
Y	0			6			12	…

- **Y** n'est défini qu'à certains instants : à l'horloge **C**
- Flot non défini est différent de nil (valeur non déterminée)

WHEN: sous-échantillonage

```
X = NAT() ;
C = (X mod 3 = 0) ;
Y = (2 * X) when C ;
```

t	t_0	t_1	t_2	t_3	t_4	t_5	t_6	…
X	0	1	2	3	4	5	6	…
2 * X	0	2	4	6	8	10	12	…
C	T	F	F	T	F	F	T	…
Y	0			6			12	…

- **Y** n'est défini qu'à certains instants : à l'horloge **C**
- Flot non défini est différent de nil (valeur non déterminée)

~~Z = X + Y ;~~

- Cette équation est illégale

WHEN: sous-échantillonage

```
X = NAT() ;
C = (X mod 3 = 0) ;
Y = (2 * X) when C ;
```

t	t_0	t_1	t_2	t_3	t_4	t_5	t_6	…
X	0	1	2	3	4	5	6	…
2 * X	0	2	4	6	8	10	12	…
C	T	F	F	T	F	F	T	…
Y	0			6			12	…

- **Y** n'est défini qu'à certains instants : à l'horloge **C**
- Flot non défini est différent de nil (valeur non déterminée)

~~Z = X + Y ;~~

- Cette équation est illégale

```
Z = current Y ;
```

Y	0			6			12	…
Z	0	0	0	6	6	6	12	…

Causalité

- Problème:

  Etablir que le programme détermine une valeur et une seule pour tous les flots.

- Solution Simple et radicale :
  - Les équations sont orientées
  - Tout flot doit avoir une et une seule définition
  - Pas de boucle qui ne passe pas par un PRE

- Justification :
  - L'exigence du domaine d'application de SCADE
  - L'exigence des utilisateurs de SCADE

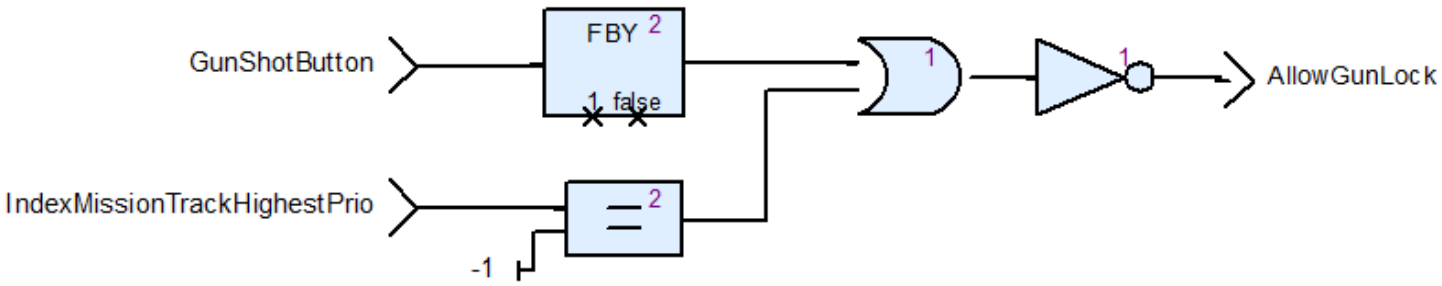
- En Pratique, c'est une trivialité :
  - Les équations définissent la relation « est définie par »
  - On vérifie que cette relation est un ordre strict

Compilation de Scade (KCG)

- Transformation du graphique en texte
- Lecture et vérification du programme:
  - La syntaxe
  - Certaines propriétés “générales”
  - Les types
  - Les horloges
  - La causalité (Scade 6)
  - L’initialisation des flots (Scade 6)
- Plongement dans le langage noyau
- Expansion (inlining)
- Simplifications (optimisations)
- Ordonnancement des équations et déclaration des mémoires
- Transformation dans un langage impératif
- ….
- Génération de code C / Ada

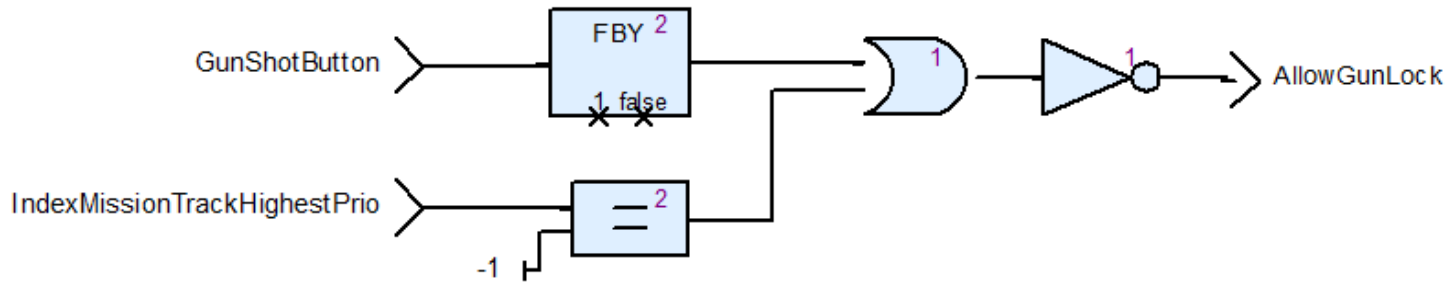
Graphique vers Texte

Toute “Boîte” est interprétée comme une équation.



Graphique vers Texte

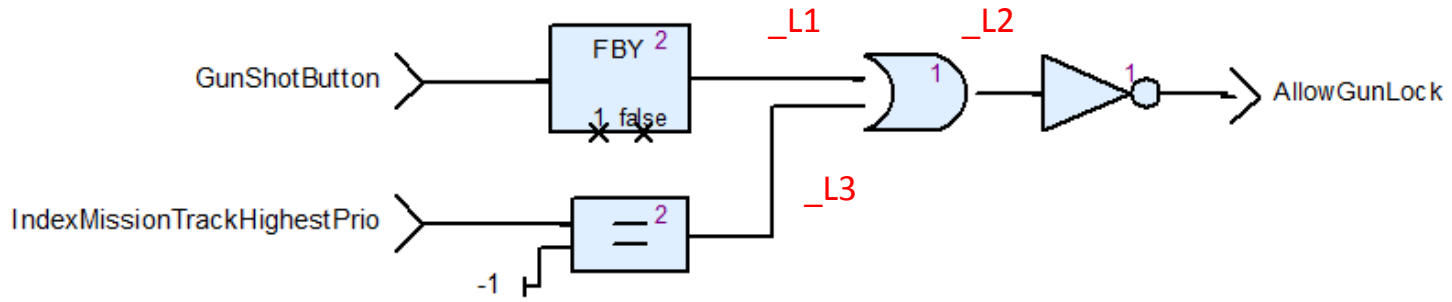
Toute “Boîte” est interprétée comme une équation.



AllowGunLock = **not** (**fby** (1; false; GunShotButton) **or (**IndexMissionTrackHighestPrio = -1)) ;

Graphique vers Texte

Toute “Boîte” est interprétée comme une équation.



AllowGunLock = **not** (**fby** (1; false; GunShotButton) **or (**IndexMissionTrackHighestPrio = -1)) ;

_L1 = **fby** (1; false; GunShotButton) ;

_L2 = _L1 **or** _L3 ;

AllowGunLock = **not** _L2 ;

_L3 = IndexMissionTrackHighestPrio = -1 ;

Compilation de Scade (KCG)

- Transformation du graphique en texte
- Lecture et vérification du programme:
  - La syntaxe
  - Certaines propriétés “générales”
  - Les types
  - Les horloges
  - La causalité (Scade 6)
  - L’initialisation des flots (Scade 6)
- Plongement dans le langage noyau
- Expansion (inlining)
- Simplifications (optimisations)
- Ordonnancement des équations et déclaration des mémoires
- Transformation dans un langage impératif
- ….
- Génération de code C / Ada

Expansion

- Objet : remplacer un nœud par ses équations
- Problème : conserver la traçabilité du code généré
- Scade : l'expansion est facultative

- Pourquoi expanser ou ne pas expanser ?
  - Performances : joue dans les 2 sens !
  - Résoudre des problèmes de causalité

Expansion et causalité

- Résoudre des problèmes de causalité

```
node F(a,b: int) returns (x,y: int)
  x , y = ( a + 1 , 0 -> pre b) ;

node Test (a:int) returns (x:int)
var
  aux : int ;
let
  aux , x = F (a, aux) ;
tel
```

```
node Test (a:int) returns (x:int)
var
  aux : int ;
  F_a, F_b, F_x, F_y : int;
let
  F_a = a ;
  F_b = aux ;
  aux = F_x ;
  x = F_y ;

  F_x, F_y = (F_a + 1, 0 -> pre F_b) ;
tel
```

Après simplification :

```
aux = a + 1 ;
x = 0 -> pre aux ;
```

Compilation de Scade (KCG)

- Transformation du graphique en texte
- Lecture et vérification du programme:
  - La syntaxe
  - Certaines propriétés “générales”
  - Les types
  - Les horloges
  - La causalité (Scade 6)
  - L’initialisation des flots (Scade 6)
- Plongement dans le langage noyau
- Expansion (inlining)
- Simplifications (optimisations)
- Ordonnancement des équations et déclaration des mémoires
- Transformation dans un langage impératif
- ….
- Génération de code C / Ada

Ordonnancement (scheduling)

- Causalité :

  Il existe un ordre sur les équations pour lequel tout flot est écrit “avant” d’être lu.

  Il en existe plusieurs et ils ne sont pas équivalents pour le code qui sera généré.

```
X = NAT() ;
Y = 1 + pre X ;
Z = X + 1 ;
```

X est causalement en relation avec Z.
Y est indépendant de X et de Z.
3 séquences sont possibles.

Ordonnancement (scheduling)

- Causalité :

  Il existe un ordre sur les équations pour lequel tout flot est écrit “avant” d’être lu.

  Il en existe plusieurs et ils ne sont pas équivalents pour le code qui sera généré.

```
X = NAT() ;
Y = 1 + pre X ;
Z = X + 1 ;
```

X est causalement en relation avec Z.
Y est indépendant de X et de Z.
3 séquences sont possibles.

```
Y = 1 + pre X ;
X = NAT() ;
Z = X + 1 ;
```

```
Y = 1 + mem_X ;
X = NAT() ;
Z = X + 1 ;
…
mem_X = X ;
```

```
Y = 1 + mem_X ;
mem_X = NAT() ;
Z = X + 1 ;
```

Compilation de Scade (KCG)

- Transformation du graphique en texte
- Lecture et vérification du programme:
  - La syntaxe
  - Certaines propriétés “générales”
  - Les types
  - Les horloges
  - La causalité (Scade 6)
  - L’initialisation des flots (Scade 6)
- Plongement dans le langage noyau
- Expansion (inlining)
- Simplifications (optimisations)
- Ordonnancement des équations et déclaration des mémoires
- Transformation dans un langage impératif
- …
- Génération de code C / Ada

Pouvoir d'expression

- Peut-on calculer ABRO ?
  - Oui … mais …

- Pourquoi vouloir étendre Scade ?
  - Solidifier encore les fondations du langage
  - Le passage à l'échelle (génie logiciel, tableaux)
  - Introduire plus de « contrôle »

Etendre Scade à quelles conditions ?

- Ne pas perdre les propriétés fondamentales :
  - Extension conservative
  - L'orientation Flot de données
  - La traçabilité du code généré
  - Les analyses de correction présentes dans Scade

Réforme de l'orthographe

- Harmoniser/simplifier la syntaxe de Scade

```
node Plus (a,b: int) returns (x: int)  ;
let
   x = a + b ;
tel
```

- Anecdotique mais :
  - Non-ambiguïté
  - Cohérence : simplicité, lisibilité, utilisabilité
  - Acceptation du langage par les utilisateurs
  - Embarqué critique implique beaucoup de revues

Vérification de l'initialisation

```
X = NAT() ;
Y = pre X ;
Z = Y -> 1 + pre Z ;
```

t	t_0	t_1	t_2	t_3	…
X	0	1	2	3	…
Y	nil	1	2	3	…
Z	nil	nil	nil	nil	…

- On accepte que certains flots soient localement non-déterminés à certains instants dans des programmes qui sont globalement déterministes.

- Un flot non-défini ne doit atteindre ni une sortie, ni une mémoire.

Vérification de l'initialisation par typage

- 2 types :
  - **0** : type des flots initialisés sans délai
  - **1** : type des flots initialisés avec un délai de 1 tick
  - avec $\mathbf{1} \sqcup \delta = \mathbf{1}$ et $\mathbf{0} \sqcup \delta = \delta$

$$\frac{e \ : \ \mathbf{0}}{\mathbf{pre}\ e \ : \ \mathbf{1}} \qquad \frac{e_1 : \delta_1 \quad e_2 : \delta_2}{e_1 \rightarrow e_2 : \delta_1} \qquad \frac{e_1 : \delta_1 \quad e_2 : \delta_2}{e_1 + e_2 : \delta_1 \sqcup \delta_2}$$

- voir J-L. Colaço, M. Pouzet : Type-based Initialization Analysis of a Synchronous Data-flow Language.

Vérification de l'initialisation par typage

- 2 types :
  - **0** : type des flots initialisés sans délai
  - **1** : type des flots initialisés avec un délai de 1 tick
  - avec $\mathbf{1} \sqcup \delta = \mathbf{1}$ et $\mathbf{0} \sqcup \delta = \delta$

$$\frac{e \ : \ \mathbf{0}}{\mathbf{pre}\ e \ : \ \mathbf{1}} \qquad \frac{e_1 : \delta_1 \quad e_2 : \delta_2}{e_1 \rightarrow e_2 : \delta_1} \qquad \frac{e_1 : \delta_1 \quad e_2 : \delta_2}{e_1 + e_2 : \delta_1 \sqcup \delta_2}$$

X = NAT() ;

Y = **pre** X ;

Z = Y **->** 1 + **pre** Z ;

$$\frac{\dfrac{X = \mathrm{NAT}() : \mathbf{0}}{\mathrm{Y} = \mathbf{pre}\ \mathrm{X} : \mathbf{1}} \qquad \dfrac{1 : \mathbf{0} \qquad \dfrac{\mathrm{Z} : \mathbf{0} = \delta}{\mathrm{pre\ Z} : \mathbf{1}}}{1 + \mathrm{pre\ Z} : \mathbf{0} \sqcup \mathbf{1} = \mathbf{1}}}{\mathrm{Z} = \mathrm{Y} \rightarrow 1 + \mathrm{pre\ Z} : \mathbf{1} = \delta}$$

- voir J-L. Colaço, M. Pouzet : Type-based Initialization Analysis of a Synchronous Data-flow Language.

Vérification de la causalité

- Vérifier qu'il n'y a pas de cycle reliant une variable à elle-même.
- Ce n'est pas satisfaisant pour les nœuds expansés.
- Vérification de la causalité par typage.

  - Voir P. Cuoq and M. Pouzet : Modular Causality in a Synchronous Stream Language. (*ESOP'01)*

Vérification de la causalité par typage

```
node F(a,b: int) returns (x,y: int)
  x , y = ( a + 1 , 0 -> pre b) ;

m , n = F (p, m) ;
```

F non-expansé : $\forall \gamma_1 \gamma_2 . \gamma_1 \times \gamma_2 \rightarrow \gamma_1 \cup \gamma_2$

F expansé : $\forall \gamma_1 \gamma_2 \gamma_3 . \gamma_1 \times \gamma_2 \rightarrow \gamma_1 \times \gamma_3$

m,n : $\gamma_m \times \gamma_n$ et p,m : $\gamma_p \times \gamma_m$

F non-expansé : m,n = F(p,m) : $\gamma_m \times \gamma_n \ < \ \gamma_p \cup \gamma_m$ illégal

F expansé : m,n = F(p,m) : $\gamma_m \times \gamma_n \ < \ \forall \gamma_3 . \gamma_p \times \gamma_3$ légal

Introduction des « Package »

- But :
  - Organiser le programme source en regroupant certaines parties.
  - Faciliter la réalisation de bibliothèques.
  - Restreindre la visibilité de certaines parties.
  - Fournir des notations pointées et des directives de visibilité.

- La vue donnée dans SCADE n'est pas ordonnée.

- On ne fixe pas de règles de masquage on interdit tout masquage ambigu (au même niveau).

Le polymorphisme

- But: similaire aux « package »
  - « Write Things Once »
  - Réutilisabilité

```
node OnlyOn (a: ‘T; c: bool) returns (x: ‘T)
let
  x = if c then a else pre x ;
tel
```

- Les contraintes ne proviennent pas de Scade mais du langage cible :
  - Pas de polymorphisme en C et le type **void*** est formellement interdit (ainsi que la plupart des transtypages).

Restriction au polymorphisme

- Un programme Scade peut contenir des parties polymorphes mais doit être monomorphe !

- En pratique, un programme Scade ne doit pas contenir de variables de type libres après typage.

Les tableaux en Scade 5

- Les tableaux existent mais pas les boucles !
- Pas d'indexation dynamique non plus.

```
node Sum (a: int^5) returns (x: int)
let
  x = a[0] + a[1] + a[2] + a[3] + a[4] ;
tel
```

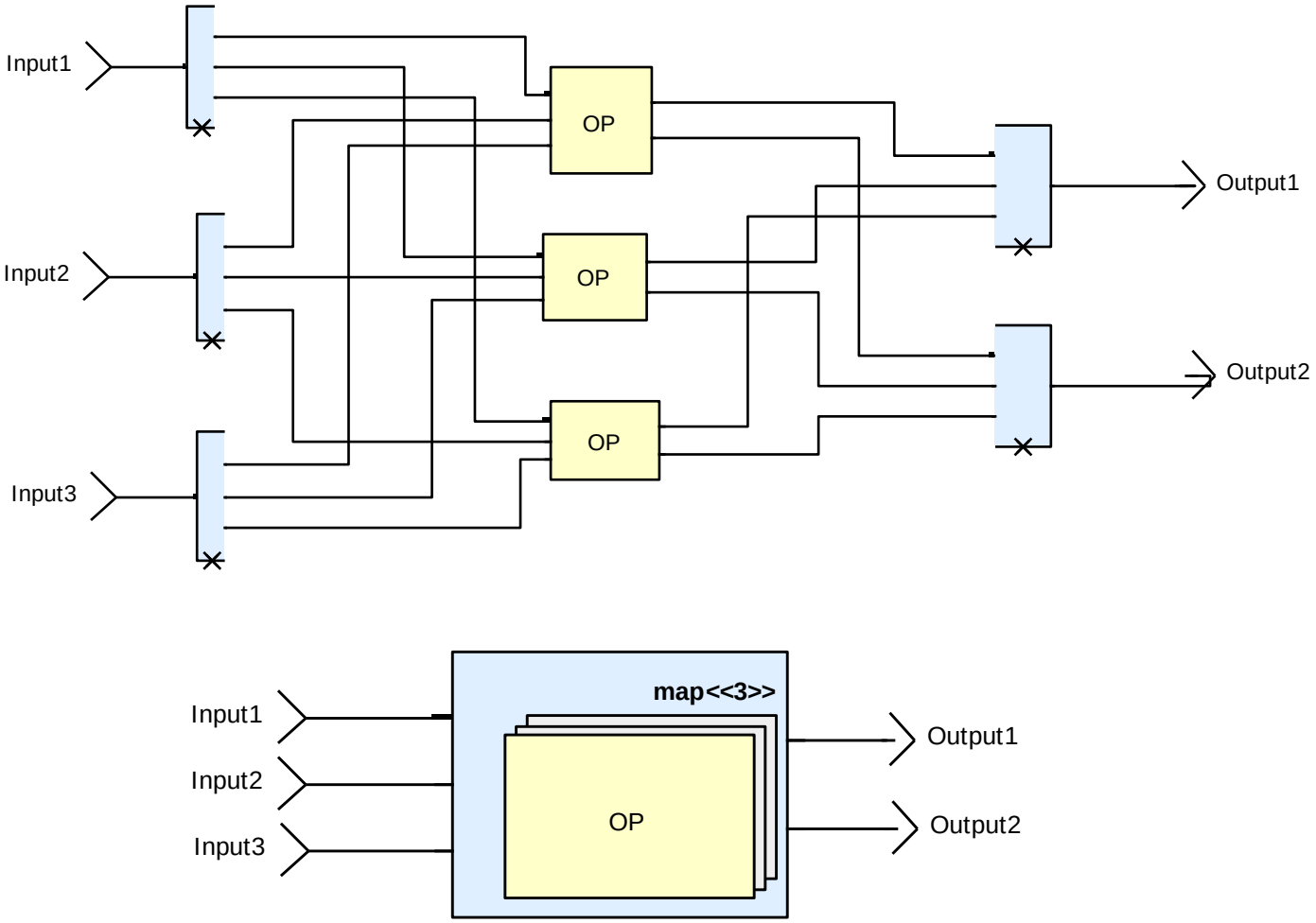
```
node VectOfInt (a: int) returns (x: int^5)
let
  x = [ a, a, a, a, a] ;
tel
```

- Manipuler un tableau de taille N nécessite un programme de taille N.

Dessiner des boucles ‘for’ ?

- Comment leur donner un sens « flot de données » ?
- Comment écrire une boucle graphiquement ?
- Comment rester dans le cadre « sur » ?
- Comment générer du code efficace et lisible ?
  - présentation à Synchron ‘99 par N. Halbwachs et J-L. Colaço;
  - thèse de L. Morel
- des combinateurs (itérateurs) d’ordre supérieur
- quelques opérateurs supplémentaires
- Indexation dynamique avec un défaut

MAP : vision flot de données



MAP

$$\text{OP} : I_1 \times \cdots \times I_p \rightarrow O_1 \times \cdots \times O_q$$

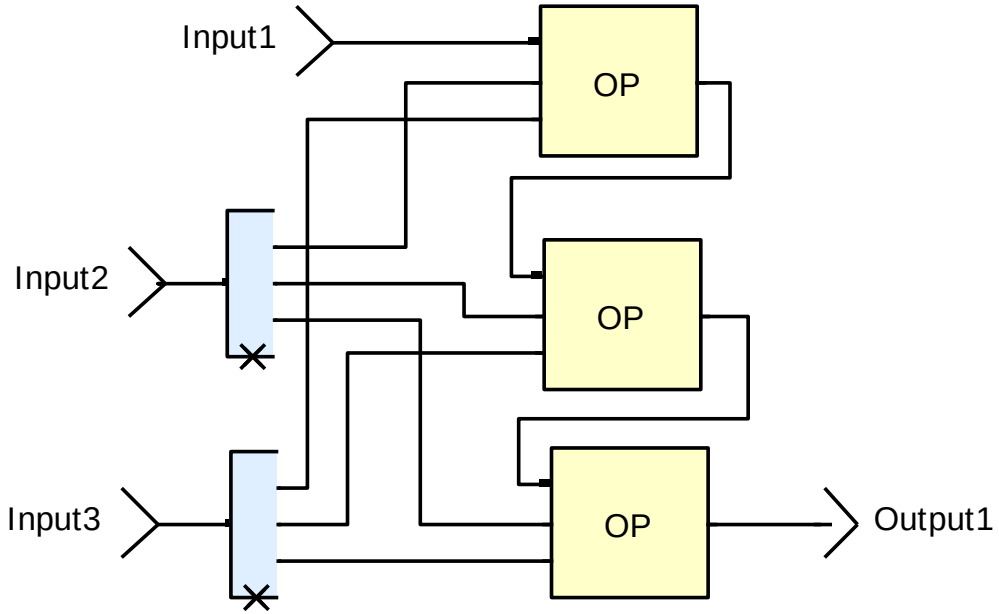
$$(\textbf{map}\ \text{Op} <<\text{n}>>) : {I_1}^n \times \cdots \times {I_p}^n \rightarrow {O_1}^n \times \cdots \times {O_q}^n$$

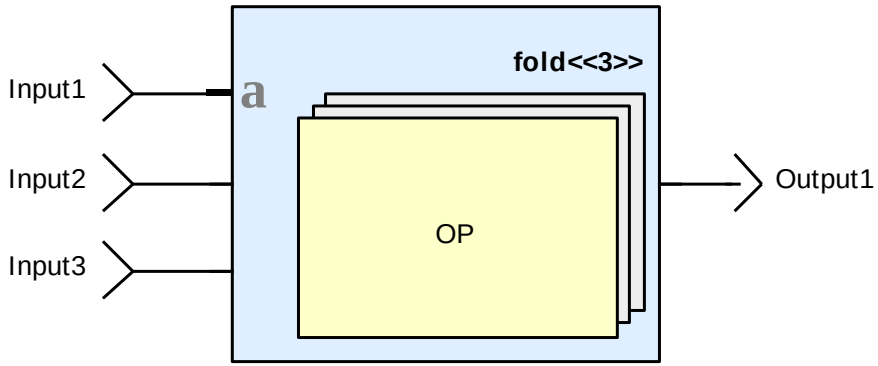
```
Y1, …, Yq = (map Op <<n>>) (X1, …, Xp) ;
```

```
Y1[0], …, Yq[0] =  Op(X1[0], …, Xp[0])
Y1[1], …, Yq[1] =  Op(X1[1], …, Xp[1])
……..
Y1[n-1], …, Yq[n-1] =  Op(X1[n-1], …, Xp[n-1])
```

```
for (i=0; i<n; i++) {
   Op (X1[i], …, Xp[i], &Y1[i], …, &Yq[i]) ;
}
```

FOLD : vision flot de données





FOLD

$$\text{OP} : \text{ACC} \times I_1 \times \cdots \times I_p \rightarrow ACC$$

$$(\textbf{fold}\ \text{Op}\ \langle\langle n \rangle\rangle) : \text{ACC} \times {I_1}^n \times \cdots \times {I_p}^n \rightarrow ACC$$

```
Z = (fold Op <<n>>) (X, Y1, …, Yp) ;
```

```
Z = Op( … (Op (Op(X, Y1[0], …, Yp[0]),  Y1[1], …, Yp[1])) … ), Y1[n-1], …, Yp[n-1]) ;
```

```
 Z= X ;
 for (i=0; i<n; i++) {
    Z = Op (Z, Y1[i], … , Yp[i]) ;
}
```

Tableaux: Autres constructions

- Mapfold, mapfoldi, mapfoldw, mapfoldwi, …

- Créer un tableau : `Z = X ^ n ;` `Z = [ X, X, …, X ] ;`

- Extraire d'un tableau : `Z = X [ n .. p ] ;` `Z = [ X[n], X[n+1], …, X[p] ] ;`

- Concaténation de tableaux : `Z = X @ Y ;` `Z = [ X[0], ..,X[n],Y[0] , …, Y[p] ] ;`

- Transposé de tableaux : `Z = transpose ( X ; n ; p ) ;`

  crée un tableau copie de X mais où les dimensions n et p sont échangées

- Indexation dynamique : `Z = ( X . [ a ] default b) ;`

- Copie avec modification : `Z = ( X with [ a ] = b) ;`

Compilation des tableaux

- Postulat :
  - obtenir des boucles ‘for’
  - ne pas casser la nature flot de données.
- Un itérateur est compilé par une boucle
- Semblable au traitement des structures.
- Causalité :
  - Restriction liée à la forme générée

Le produit de deux matrices : Produit scalaire

```
function prod_sum (a,v,w:  real) returns (aa: real)
  aa = a + v*w;

-- scalar product of two vectors: V . W
function ScalProd (V,W: real^n) returns (sp: real)
  sp = (fold prod_sum <<n>>) (0.0, V, W);
```

```
 /* prod_sum */
kcg_real prod_sum(
   /* prod_sum::a */ kcg_real a,
   /* prod_sum::v */ kcg_real v,
   /* prod_sum::w */ kcg_real w)
{
   /* prod_sum::aa */ kcg_real aa;

   aa = a + v * w;
   return aa;
}
```

```
 /* ScalProd */
kcg_real ScalProd(
   /* ScalProd::V */ array_real_3 *V,
   /* ScalProd::W */ array_real_3 *W)
{
   kcg_int i;
   /* ScalProd::sp */ kcg_real sp;

   sp = 0.0;
   for (i = 0; i < 3; i++) {
      sp = /* prod_sum */ prod_sum(sp, (*V)[i], (*W)[i]);
   }
   return sp;
}
```

Le produit de deux matrices : matrice x vecteur

```
-- product of a matrix by a vector: A * u
function MatVectProd (A: real^m^n; u: real^n)  returns  (w: real^m)
   w = (map ScalProd <<m>>)(transpose (A; 1; 2), u^m);
```

```
 /* MatVectProd */
void MatVectProd(
   /* MatVectProd::A */ array_real_4_3 *A,
   /* MatVectProd::u */ array_real_3 *u,
   /* MatVectProd::w */ array_real_4 *w)
{
  array_real_3_4 tmp;
  kcg_int i1;
  kcg_int i;

  for (i = 0; i < 3; i++) {
     for (i1 = 0; i1 < 4; i1++) {
        tmp[i1][i] = (*A)[i][i1];
     }
  }
  for (i = 0; i < 4; i++) {
     (*w)[i] = /* ScalProd */ ScalProd(&tmp[i], u);
  }
}
```

Le produit de deux matrices

```
-- matrix product: A * B  (  (m,n) x (n,p)  )
function MatProd (A:real^m^n; B:real^n^p) returns (C:real^m^p)
   C = (map MatVectProd <<p>>)(A^p, B);
```

```
 /* MatProd */
void MatProd(
   /* MatProd::A */ array_real_4_3 *A,
   /* MatProd::B */ array_real_3_5 *B,
   /* MatProd::C */ array_real_4_5 *C)
{
  kcg_int i;

  for (i = 0; i < 5; i++) {
     /* MatVectProd */ MatVectProd(A, &(*B)[i], &(*C)[i]);
  }
}
```

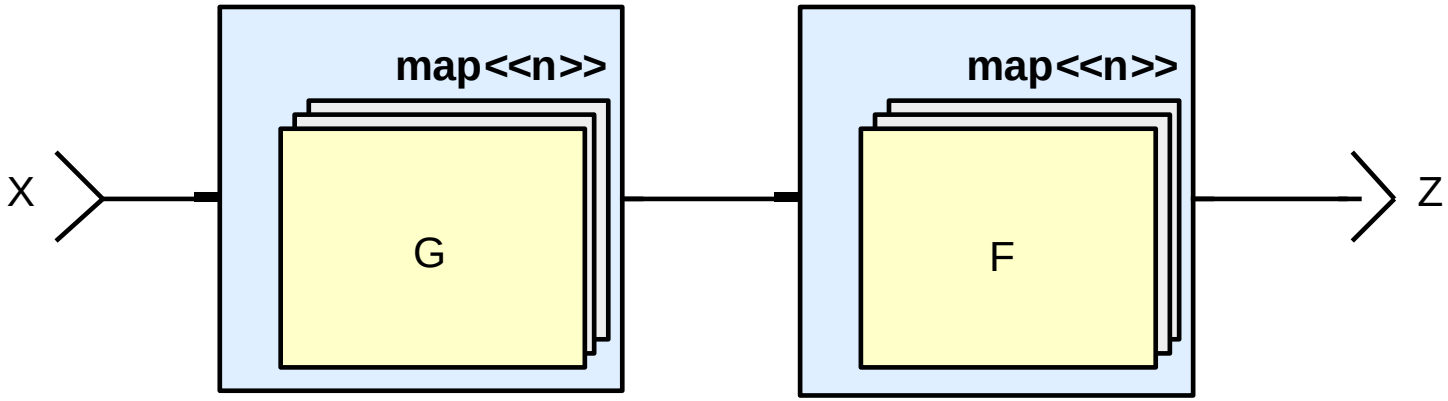
Optimisation des tableaux

- Simplifier des identités remarquables :
  - (map F <<n>>) ((map G <<n>>) (X))

    = (map F <<n>>) ([G(X[0]) , … , G(X[n-1])])

    = [F(G(X[0])) , … , F(G(X[n-1]))]

    = (map (F o G) <<n>>) (X) ;



Optimisation des tableaux

- Ne pas construire les tableaux seulement itérés

```
Y = (map F <<n>>) ( X ^ n) ;
```

```
for (i = 0; i < n; i++) {
     Y = F (X) ;   }
}
```

Le produit de deux matrices : expansé

```
 /* MatProd */
void MatProd(inC_MatProd *inC, outC_MatProd *outC)
{
  kcg_int i;
  kcg_int i_ScalProd_MatVectProd;
  kcg_int i_MatVectProd;

  for (i = 0; i < 5; i++) {
    for (
      i_MatVectProd = 0;
      i_MatVectProd < 4;
      i_MatVectProd++)  {
      outC->C[i][i_MatVectProd] = 0.0;
     for (
       i_ScalProd_MatVectProd = 0;
       i_ScalProd_MatVectProd < 3;
       i_ScalProd_MatVectProd++)  {
       outC->C[i][i_MatVectProd] = outC->C[i][i_MatVectProd] +
            inC->A[i_ScalProd_MatVectProd][i_MatProd_MatVectProd] *
            inC->B[i][i_ScalProd_MatVectProd];
     }
    }
  }
}
```

Le produit de deux matrices : expansé

```
 /* MatProd */
void MatProd(inC_MatProd *inC, outC_MatProd *outC)
{
  kcg_int i;
  kcg_int j;   /* i_ScalProd_MatVectProd; */
  kcg_int k;  /* i_MatVectProd; */

  for (i = 0; i < 5; i++) {
    for (k = 0; k < 4; k++) {
       outC->C[i][k] = 0.0;
       for (j = 0; j < 3;j++) {
         outC->C[i][k] = outC->C[i][k] + inC->A[j][k] * inC->B[i][j];
       }
     }
   }
}
```

Demeurent des améliorations à faire

- Copie de tableaux
  - voir A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler.
- Fusion des boucles dans la partie impérative

Plus de Contrôle

- Contrôle : choisir de ne pas faire un calcul
  - Performances (à relativiser WCET)
  - Le Calcul n'est pas toujours possible

- Dans un contexte flot de données
  - Lois déterminées par des plages de fonctionnement
  - Programme ronronnant mais avec des discontinuités

- En Scade : une seule construction **when** : Spartiate !

- En Scade 6 :
  - À la Synchart (Statechart – Stateflow®)
  - Mélange complet (Simulink-Stateflow ou Esterel-Synchart)

Se débarasser de **current**

- **Current** : incompatible avec l'analyse d'initialisation

t	t_0	t_1	t_2	t_3	t_4	t_5	t_6	…
X = NAT()	0	1	2	3	4	5	6	…
Y = X when X mod 3 = 0	0			3			6	…
current Y	0	0	0	3	3	3	6	…

Se débarasser de **current**

- **Current** : incompatible avec l'analyse d'initialisation

t	t_0	t_1	t_2	t_3	t_4	t_5	t_6	…
X = NAT()	0	1	2	3	4	5	6	…
Y = X when X mod 3 = 0	0			3			6	…
Z = X when X mod 3 = 2			2			5		
current Y	0	0	0	3	3	3	6	…
current Z	nil	nil	2	2	2	5	5	…

Se débarrasser de **current**

- **Current** : incompatible avec l'analyse d'initialisation

t	t_0	t_1	t_2	t_3	t_4	t_5	t_6	…
X = NAT()	0	1	2	3	4	5	6	…
Y = X when X mod 3 = 0	0			3			6	…
Z = X when X mod 3 = 2			2			5		
W = X when IN								
current Y	0	0	0	3	3	3	6	…
current Z	nil	nil	2	2	2	5	5	…
current W	nil	nil	nil	nil	nil	nil	nil	…

Le remplacer par **merge**

- **merge** :
  - fusion de 2 flots sur des horloges complémentaires.

X = NAT()	0	1	2	3	4	5	6	…
Y = X when IN		1				5		…
Z = X when not IN	0		2	3	4		6	
Merge (IN; Y; Z)	0	1	2	3	4	5	6	

- Équivaut à If IN then current Y else current Z
  - Mais vérifiable statiquement (moins de bug)
  - Ne pose aucun problème pour l'analyse d'initialisation

- Les horloges sont étendues aux énumérés

Programmer des modes

- Pbm : Selon une condition **C(IN)**, on a deux lois **F1** et **F2** pour calculer **OUT**.

- En utilisant **merge** :

```
OUT1 = F1 (IN when cond) ;
OUT2 = F2 (IN when not cond) ;
OUT = merge (IN; OUT1; OUT2) ;
cond = C(IN) ;
```

  - Il est nécessaire de :
    - Déclarer les différentes composantes de chaque variable
    - D'associer les horloges à toutes les variables
  - Ca devient complexe quand :
    - il y a beaucoup d'équations
    - il y a des sous modes imbriqués

Programmer des modes

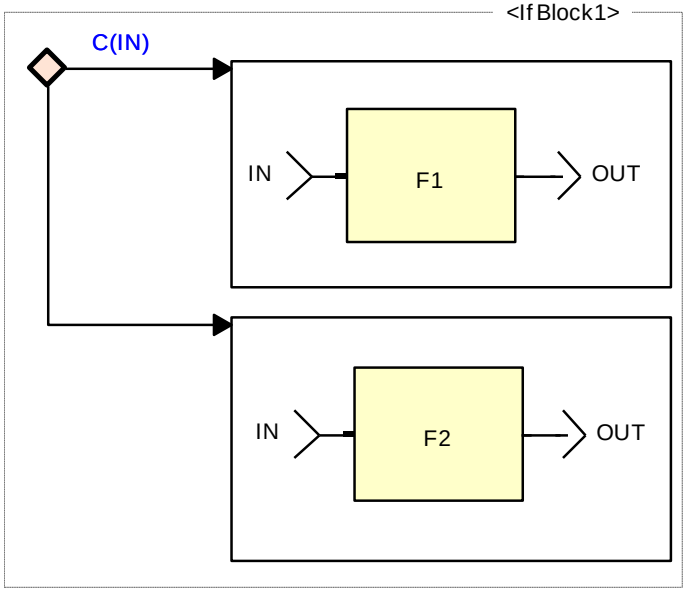
- Pbm : Selon une condition **C(IN)**, on a deux lois **F1** et **F2** pour calculer **OUT**.

- En utilisant **merge** :

```
OUT1 = F1 (IN when cond) ;
OUT2 = F2 (IN when not cond) ;
OUT = merge (IN; OUT1; OUT2) ;
cond = C(IN) ;
```

- On se dote d'une syntaxe :

```
activate if C(IN) then
   OUT = F1 (IN) ;
else
   OUT = F2 (IN) ;
returns OUT;
```



Compiler les modes

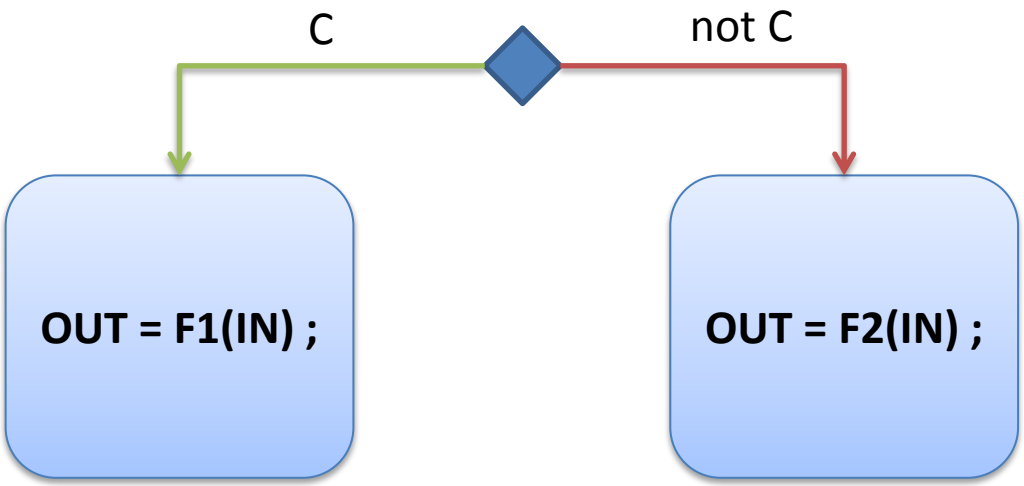
- La compilation des modes suit exactement le schéma de l'exemple précédent :

```
activate if C(IN) then
   OUT = F1 (IN) ;
else
   OUT = F2 (IN) ;
returns OUT;
```

```
OUT1 = F1 (IN when cond) ;
OUT2 = F2 (IN when not cond) ;
OUT = merge (IN; OUT1; OUT2) ;
cond = C(IN) ;
```

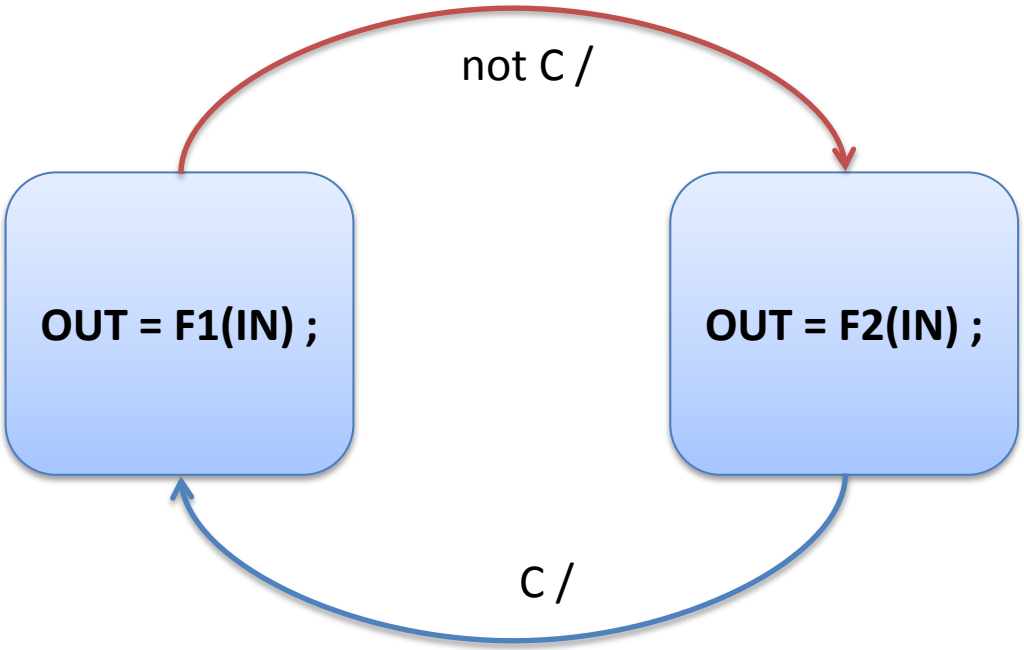
- Les modes s'expriment dans le langage noyau :
  - Extension conservative
  - Entièrement flot de données

Automate à la **mode**



- La condition est extérieure aux modes.
- Le mode actif est choisi par la condition.

Automate à la mode



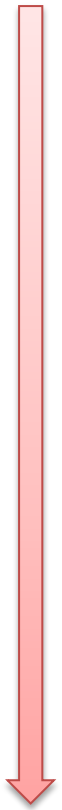
- L'automate est par défaut dans un mode.
- On choisit de demeurer ou de quitter ce mode.
- Quand quitter un mode :
  - Immédiatement (préemption forte)
  - À l'instant d'après (préemption faible)

Compiler des automates

- Informations nécessaires (à chaque instant) :
  - État sélectionné
  - Transition forte tirée
  - État actif
  - Transition faible tirée
  - Prochain état sélectionné

Compiler des automates

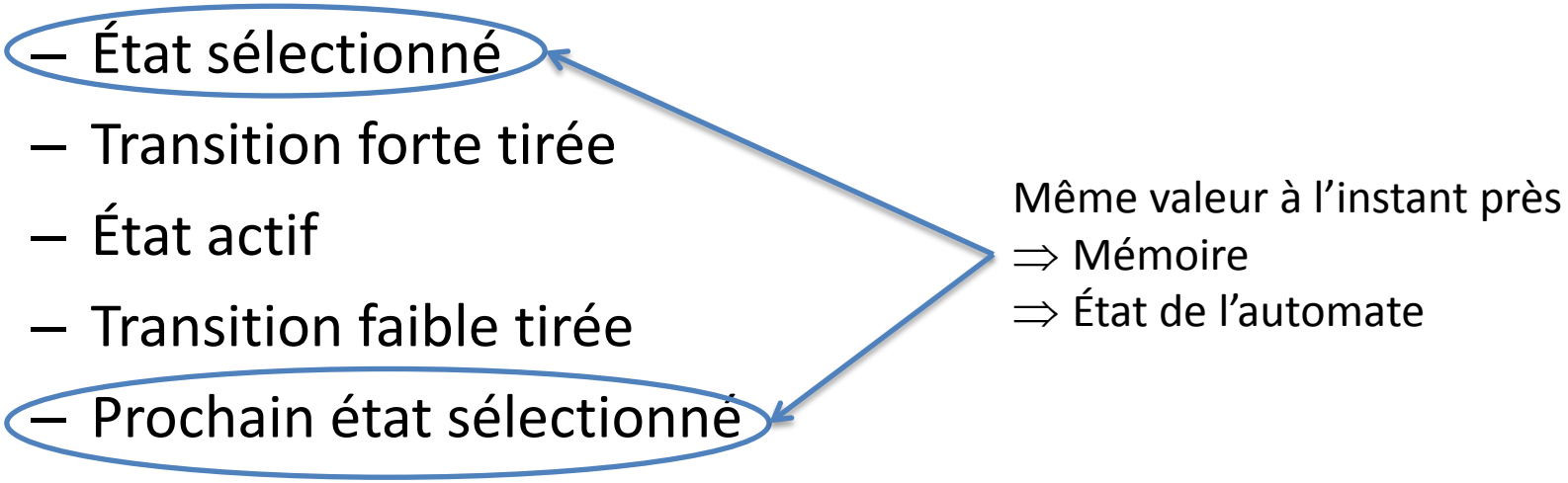
- Informations nécessaires (à chaque instant) :
  - État sélectionné
  - Conditions des transitions fortes
  - Transition forte tirée
  - Action de la transition tirée
  - État actif
  - Corps de l'état actif
  - Transition faible tirée
  - Action de la transition tirée
  - Prochain état sélectionné



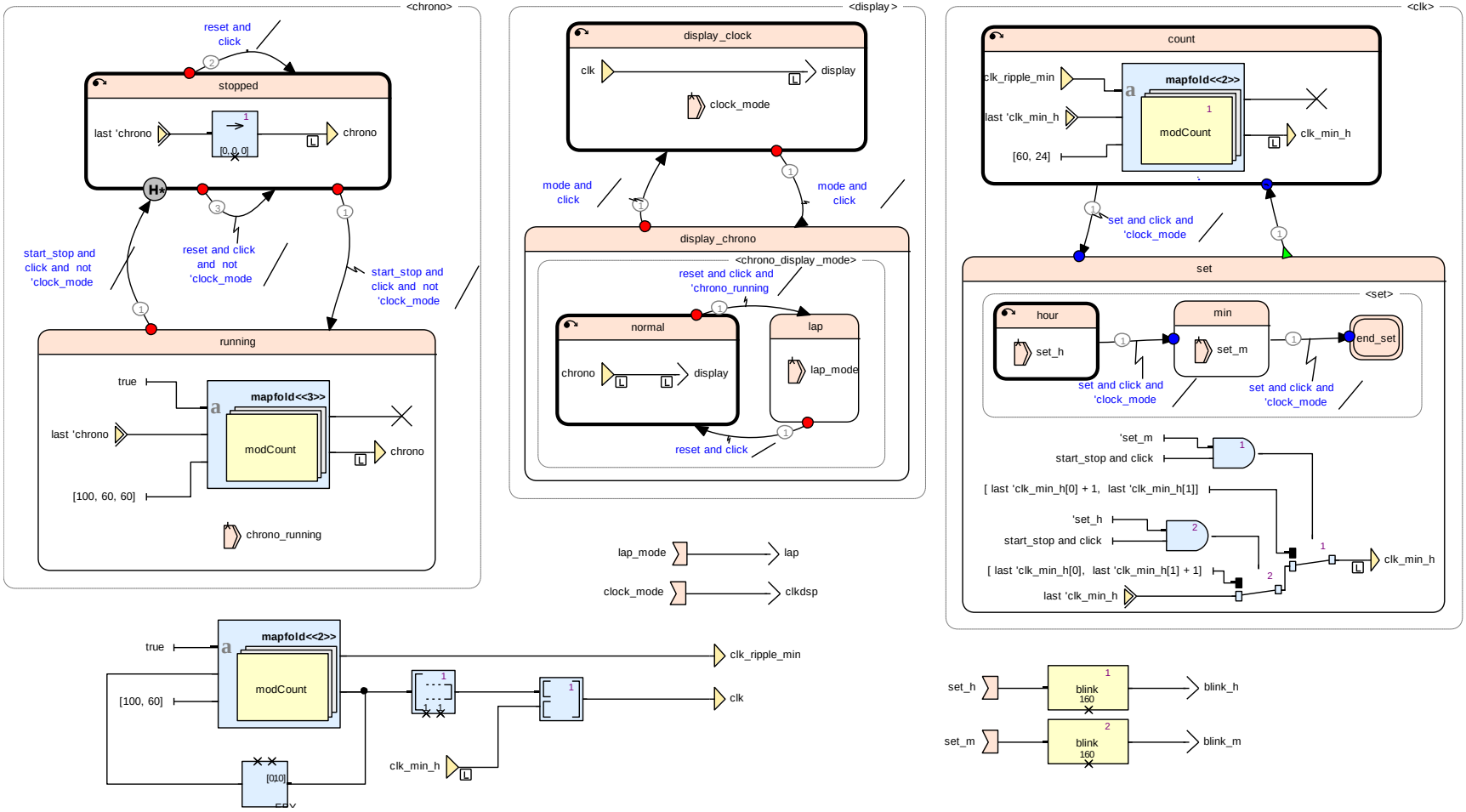
Pas un ordre d'évaluation
Mais une dépendance causale

Compiler des automates

- Informations nécessaires (à chaque instant) :
  - État sélectionné
  - Transition forte tirée
  - État actif
  - Transition faible tirée
  - Prochain état sélectionné



La montre digitale : version Scade 6



Objective Caml

- KCG est développé dans les mêmes conditions que du code destiné à être embarqué.

- Quel langage de programmation utiliser ?

- OCAML :
  - Avec une runtime simplifiée
  - Un outil de mesure de couverture

Voir Experience Report: Using Objective Caml to develop safety-critical embedded tool in certification framework.

Références

- Type-based Initialization Analysis of a Synchronous Data-flow Language.
  Jean-Louis Colaço and Marc Pouzet. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004

- Modular Causality in a Synchronous Stream Language.
  Pascal Cuoq and Marc Pouzet. (*ESOP'01)*

- Exploitation des structures régulières et des spécifications locales pour le développement correct de systèmes réactifs de grande taille.
  L. Morel, thèse de doctorat, 2005

- A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler.
  Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. *(LCTES'12)*

- Experience Report: Using Objective Caml to develop safety-critical embedded tool in certification framework.
  B. Pagano, O. Andrieu, T. Moniot, B. Canou, E. Chailloux, P. Wang, P. Manoury, J-L. Colaço. *(ICFP'09)*

Casting du Projet Scade 6 - KCG 6

- J-L. Colaço
- M. Pouzet
- F.X. Fornari

- N. Rostaing-Schmidt
- Y. Larvor
- O. Andrieu
- T. Moniot
- J. Verlaguet
- T. Rivière
- J. Forget

- G. Berry
- B. Dion

- F.X. Dormoy
- G. Collard
- L. Pouchan
- G. Maillard
- S.S. Lee

- Pardon à ceux que j'oublie