# AltaRica2Lustre

Rémi Gobard, Alain Griffault, Gérald Point
LaBRI/MVTsi

Second AltaRica Workshop

Marseille, 22 et 23 octobre 2003

# Le projet

# Les objectifs

1. Générer automatiquement un source Lustre à partir d'un source AltaRica.

   **entrée** : source, source mise à plat, modèle.

   **sortie** : source, format ec, format oc.

   **milieu** : traduction syntaxique *pure*.
2. Restreindre au minimum le langage AltaRica.
3. Être compatible avec AltaRica DF d'Antoine.
4. Générer un code lustre lisible (modifiable).

## La méthode retenue

1. Partir des analyseurs de Gérald.
2. Ajouter les modifications d'AltaRica DF une à une.
3. Traiter les rubriques une à une par l'expérimentation.

## Les outils retenus

Un prototype

1. Langage Java
2. Analyseur lexical JFlex : package Java
3. Analyseur syntaxique BYaccJ : programme C générant des sources Java.

puis intégration dans les altatools

# Les résultats obtenus

1. Un sous langage AltaRica proche d'AltaRica DF pour lequel :
   - Toute description déterministe est compilée en un programme lustre sémantiquement équivalent.
   - Pour toute description non déterministe, on génère au choix :
     - soit un programme lustre acceptant (choix *arbitraire*) ;
     - soit un programme lustre non acceptant (erreur à l'exécution).
2. Un outil utilisable qui produit :
   - un source lustre lisible et syntaxiquement correct.
   - des messages explicites pour les sources AltaRica non valides.
   - un source lustre dont les erreurs à l'exécution sont sémantiquement explicables.

# Syntaxe AltaRica

```
const c1 = ...;
domain d1 = ...;
node ...
  sub    n1,..., nn : nomDeComposant;
  flow   f1,..., fn : nomDeDomaine;
  state  s1,..., sn : nomDeDomaine;
  event  e1,..., {ek} < en < {ei,ej};
  trans  gi1 |- ei -> [sj := ...,];
         gi2 |- ei -> [sj := ...,];
  assert exp1(f1,...,fn,s1,...,sn,n1.f1i,...,nn.fni);
         ...;
         expm(f1,...,fn,s1,...,sn,n1.f1i,...,nn.fni);
  sync   <[ei],...,[nj.ejk],...>;
         <[ei],...,[nj.ejk]?,...>;
         <[ei],...,[nj.ejk]?,...> op nb;
  extern initial_state = s1 = ..., [sj = ...,] ..., sn = ...;
edon
```

# AltaRica DF / AltaRica

**Pour une feuille :**

- Les flux sont orientés : in|out|local.
- Il n'y a plus de priorités sur les événements.
- La visibilité est globale pour les flux, états et événements.
- Les assertions sont réduites à :
  `fli = exp(fi1,...,fin,fl1,...,fln,s1,...,sn);`
  `foi = exp(fi1,...,fin,fl1,...,fln,s1,...,sn);`
  et unicité des occurrences des `fl` et`fo`.
- Une clause `init` est possible, pour les états.

**Pour un nœud :**

- Les flux sont orientés : in|out|local.
- Il n'y a plus d'états et d'événements déclarés.
- Les synchronisations ne permettent pas la diffusion.
- Les synchronisations déclarent les événements.
- Les assertions sont réduites à : `fiIn = fjOut`

## Les avantages d'AltaRica DF

1. Les composants sont des fonctions séquentielles.
2. La compilation vers les formules booléennes est efficace.
3. Il formalise le dialecte AltaRica Dassault-France.
4. Il répond aux besoins de description des systèmes ayant de nombreux composants reliés par des flux, et dont les seules évolutions sont les pannes (et les réparations).
5. Il est sémantiquement compatible avec AltaRica.

## Les défauts d'AltaRica DF

1. Restriction importante.
2. Quelques choix par défaut différents de ceux d'AltaRica.

Le processus de traduction du typage

# Les identificateurs

**AltaRica**

– `[a-z,A-Z,_][a-z,A-Z,0-9,_]*` pour les déclarations.
– `identInstanceNoeud.ident` pour l'utilisation des flux et des événements.
– `ident` pour l'utilisation des états.
– le caractère "." représente la hiérarchie.

**AltaRica DF**

– `[a-z,A-Z,_][a-z,A-Z,0-9,_]*` pour les déclarations.
– `(identInstanceNoeud.)*ident` pour l'utilisation
– le caractère "." représente la hiérarchie (surcharge).

**Lustre**

– `[a-z,A-Z,_][a-z,A-Z,0-9,_]*` pour les déclarations.
– la hiérarchie est obtenue par appels de fonctions.

# La hiérarchie

**AltaRica**

```
node A edon
node B sub a1, a2 : A; edon  // instances nommées
node C sub a1, a2 : A; edon  // instances nommées
node D sub b : B, c : C; edon
```

**Lustre**

```
node A(...) returns (...);
let ... tel
node B(...) returns (...);
let a1 = A(...); a2 = A(...); tel
node C(...) returns (...);
let a1 = A(...); a2 = A(...); tel
node D(...) returns (...);
let b = B(...); c = C(...); tel
```

# La traduction des identificateurs

C'est la mise à plat classique par concaténation des identificateurs.

**Un nœud quelconque**

1. Il définit un nœud de même nom,
2. les identificateurs des flux sont préfixés par `f_`, des états par `s_` et des événements (hors synchro) par `e_`,
3. les identificateurs des sous-nœuds deviennent `nomSousNoeud`.

# La traduction des identificateurs (remarques)

Le principal avantage de ce codage est de conserver la surcharge du `.`. Ainsi, l'identificateur AltaRica `b.c.on` du nœud `A` désigne aussi bien :

1. la variable `b.c.on` de `A`,
2. la variable `c.on` du sous-nœud `b`,
3. la variable `on` du sous-nœud `c` du sous-nœud `b`.

Les défauts majeurs :

1. Le code lustre peutt être incorrect en présence de `_`.
2. Du fait qu'un événement devient une variable (seul choix possible), le code lustre peut être incorrect si un événement porte le même nom qu'un flux ou bien un état.

C'est l'explication du préfixage.

# La traduction des identificateurs (un exemple)

**Lustre généré**

```
node CircuitV1(G_failure_, G_repair_, S_push_, L_reaction_, __L___S___G_
returns (G_s_on, S_s_on, L_s_on : bool);
var
  G_f_f1, G_f_f2, S_f_f1, S_f_f2, L_f_f1, L_f_f2 : bool;
let
  G_s_on = Generator(G_f_f1, G_f_f2);
  S_s_on = OrientedSwitch(S_f_f1, S_f_f2);
  L_s_on = LamplightV1(L_f_f1, L_f_f2);
  S_f_f1 = G_f_f1;
  L_f_f1 = S_f_f2;
  L_f_f2 = G_f_f2;
tel
```

# Les constantes et les domaines

**Les constantes**

**AltaRica :** `enum, exp_bool, exp_num`

**Lustre :** `exp_bool, exp_num, exp_reelle`

**Les domaines**

**AltaRica :** `bool, [int1,int2], {cste1, cste2, ..., csten}`

**Lustre :** `bool, integer, real`

# La traduction des constantes et les domaines

**les constantes booléennes et entières** sont recopiées à l'identique.

**les constantes énumérées** deviennent soient des constantes globales entières (cas des constantes globales), soient des variables locales entières. Le traitement est **global** par association avec un entier unique.

**les variables booléennes** sont inchangées.

**les variables dans un intervalle entier** deviennent des variables entières contraintes par des assertions sur les bornes ($\geq$ Binf et $\leq$Bsup).

**les variables dans un domaine énuméré** deviennent des variables entières contraintes par des assertions (cste1 or cste2 or ... or csten).

Un problème :

– La constante énumérée est globale en AltaRica. On peut tester l'égalité et l'inégalité.

– Les constantes globales en Lustre peuvent être masquées localement.

$\Rightarrow$ Le code généré (qui respecte la sémantique AltaRica) peut ne pas être réutilisable tel quel du fait de ce traitement global.

# Orientation et visibilité

**AltaRica**

| | flux | | état | | événement | |
|---|---|---|---|---|---|---|
| | orientation | visibilité | orientation | visibilité | orientation | visibilité |
| AltaRica | - | père | - | private | - | père |
| AltaRica DF | in\|out\|local | global | - | global | - | global |
| Alta2Lustre | ? | ? | ? | ? | ? | ? |

**Lustre**

```
node funct (in1, ..., inp ) returns (out1, ..., outm)
var local1, ..., localn;
let
...
assert
...
tel
```

# Orientation et visibilité (les flux)

**local** désigne un calcul auxiliaire aussi bien en AltaRica DF qu'en Lustre, et il était prévu en AltaRica. → aucun problème.

**père** désigne un flux qui sort du composant, c'est la sémantique AltaRica.

**global** désigne en AltaRica DF un flux qui traverse toute la hiérarchie. C'est une sémantique facile à implémenter en AltaRica :
- explicitement en créant à chaque niveau hiérarchique un flux égal. (pattern AltaRica)
- implicitement par la surcharge de la notation pointée.

Nous avons retenu (après anglicisation) :

1. parent : in, parent : out, public : in, public : out, private.
2. traitement du public par surcharge du `.` sans duplication de variable.

# Orientation et visibilité (les états)

**AltaRica** est compositionnel, notamment du fait que les variables d'états sont locales.

**AltaRica DF** reste compositionnel malgré ses variables d'états globales, du fait que les nœuds n'ont ni état ni transition.

Les solutions possibles :

1. AltaRica $\rightarrow$ les états sont locaux.
2. AltaRica DF $\rightarrow$ les états sont globaux, mais il n'y a plus de comportement dans les nœuds.
3. États globaux et comportements dans les nœuds, mais les états des sous-nœuds ne peuvent être utilisés syntaxiquement que dans les gardes et assertions.
4. État public $\equiv$ (Flux out : public = État private) (pattern AltaRica)

**Nous avons retenu la troisième solution.**

# Orientation et visibilité (les événements)

**local** désigne un événement *interne*. C'est un $\epsilon$ nommé (réaction). C'est une sémantique implémentable en AltaRica par une boite englobante.

**père** désigne un événement qui sort du composant. Si le père ne le synchronise pas, il devient $\epsilon$. C'est la sémantique AltaRica.

**global** désigne en AltaRica DF un événement qui traverse la hiérarchie sans être synchronisé, mais en conservant son identité. C'est une sémantique facile à implémenter en AltaRica (mot réservé `equipment` rubrique `extern`) :
– explicitement en créant à chaque niveau hiérarchique un événement égal.
– implicitement par la surcharge de la notation pointée.

**Problème :** Que devient un événement `public` synchronisé ?

Nous avons retenu (après anglicisation) :

1. private : pas encore implémenté
2. parent : C'est la sémantique classique.
3. public : l'événement *traverse* la hiérarchie tant qu'il n'est pas *capturé* par une synchronisation explicite. On utilise la surcharge du `.`, et on vérifie les *droits*.

# Le langage AltaRica2Lustre : les identificateurs

- `[a-z,A-Z][a-z,A-Z,0-9]*` pour les déclarations.
- `(identInstanceNoeud.)*ident` pour l'utilisation dans les gardes et assertions.
- `ident` pour l'utilisation des états dans les affectations.
- le caractère `.` represente la hiérarchie (surcharge).

# Le langage AltaRica2Lustre : la syntaxe

```
const c1 = ...;  domain d1 = ...;
node ...
  sub    n1,..., nn : nomDeComposant;
  flow   fi1,..., fin : nomDeDomaine : public|parent : in;
         fo1,..., fon : nomDeDomaine : public|parent : out;
         fl1,..., fln : nomDeDomaine : private;
  state  s1,..., sn : nomDeDomaine : public|parent|private;
  event  e1,..., en : public|parent|private;
  trans  gi1 |- ei -> [sj := ...,];
         gi2 |- ei -> [sj := ...,];
  assert fli = exp(fi1,...,fin,fl1,...,fln,s1,...,sn,n1.fioj,...,nn.fioj
         foi = exp(fi1,...,fin,fl1,...,fln,s1,...,sn,n1.fioj,...,nn.fioj
         expi(f1,...,fn,s1,...,sn,n1.fioj,...,nn.fioj);
  sync   <[ei],...,[nj.ejk],...>;
  init   [sj := ...,];
  extern initial_state = s1 = ..., [sj = ...,] ..., sn = ...;
edon
```

# La traduction du type d'une feuille

**les paramètres** comprennent :

- les variables de flux `in`,
- les événements `parent` ou `public`.
- une variable `noinput` si les ensembles précédents sont vides.

**le résultat** comprend :

- les variables de flux `out`,
- les variables d'états `parent` ou `public`.
- une variable `noreturn` si les ensembles précédents sont vides.

**les variables** comprennent :

- les variables de flux `private`,
- les variables d'états,
- les événements `private`.

**Remarque :** On aurait peut-être pu choisir de mettre les états dans les paramètres. Les boites lustre seraient alors purement fonctionnelles (sans mémoire), mais sans doute illisibles.

# La traduction du type d'un nœud

**les paramètres** comprennent :

- les variables de flux `in`,
- les variables de flux `in:public` des sous-nœuds,
- les événements `parent` ou `public`, éventuellement renommés et dupliqués.
- les événements `public` *non capturés* des sous-nœuds.

**le résultat** comprend :

- les variables de flux `out`,
- les variables d'états `parent` ou `public`.
- les variables de flux `out:public` des sous-nœuds,
- les variables d'états `public` des sous-nœuds,
- une variable `noreturn` si les ensembles précédents sont vides.

**les variables** comprennent :

- les variables de flux `private`,
- les variables d'états `private`,
- les événements `private`.

Le processus de traduction du comportement

# Identificateur d'un événement

**private** variable locale de même nom.

**père** Ils subissent le traitement habituel des identificateurs puis :
- sont concaténés avec `epsilon` si non synchronisés.
- prennent le nom *formaté* des vecteurs de synchronisations sinon.

**public** Ils subissent le traitement habituel des identificateurs sauf s'ils sont *capturés*. Ils prennent alors le nom *formaté* des vecteurs de synchronisations, et ne sont plus accessibles par la notation *pointée*.

# La traduction des transitions (idée)

```
trans
  expBooli |- evt1 -> s1 := exp1;
  expBoolj |- evt2 -> s2 := exp2;
  expBoolk |- evt1 -> s1 := exp3; s2 := exp4;
-----------------------------------------
var guard_0, guard_1, guard_2 : bool;
let
  guard_0 = false -> pre(expBooli) and evt1;
  guard_1 = false -> pre(expBoolj) and evt2;
  guard_2 = false -> pre(expBoolk) and evt1;
  s1 = s1Init -> if guard_0 then exp1
                 else if guard_2 then exp3
                 else pre(s1); //epsilon
  s2 = s2Init -> if guard_1 then exp2
                 else if guard_2 then exp4
                 else pre(s2); //epsilon
tel
```

# La traduction des transitions (les problèmes)

1. Qu'est-ce qu'un événement ?
2. Les affectations correspondent elles aux mêmes transitions ?
3. Les valeurs initiales ?
4. Que faire en cas de non-déterminisme ?
5. Post-condition non satisfaite ?
6. Tous les événements d'une synchronisation ne sont pas possibles ?
7. La post-condition d'une synchronisation n'est pas satisfaite ?

# Qu'est-ce qu'un événement ?

**bool : niveau** : `pre(expBool) and evt;`

**Avantages :** Facile et respecte la simulation.

**Défauts :** L'événement a une durée (celle du cycle).

**bool : fronts** : `pre(expBool) and (evt <> pre(evt));`

**Avantages :** Respecte la sémantique AltaRica.

**Défauts :** Peu intuitif en Lustre.

**bool : front montant** : `pre(expBool) and (evt and not pre(evt));`

**Avantages :** Respecte la sémantique AltaRica.

**Défauts :** On ne peut pas répéter immédiatement l'événement.

**integer** : `pre(expBool) and (evt = pre(evt)+1);`

**Avantages :** Respecte la sémantique AltaRica, et permet de compter.

**Défauts :** Engendre des automates infinis.

Nous avons retenu le premier choix.

## Les affectations correspondent elles aux mêmes transitions ?

```
trans
  expBooli |- evt1 -> s1 := exp1;
  expBoolk |- evt1 -> s2 := exp2;
-----------------------------------------
var guard_0, guard_1 : bool;
let
  guard_0 = false -> pre(expBooli) and evt1;
  guard_1 = false -> pre(expBoolj) and evt2;
  s1 = s1Init -> if guard_0 then exp1
                 else pre(s1); //epsilon
  s2 = s2Init -> if guard_1 then exp2
                 else pre(s2); //epsilon
tel
```

Pour `guard_0 & guard_1`, les deux affectations sont faites (Erreur).

# Les affectations correspondent elles aux mêmes transitions (solution)

```
trans
  expBooli |- evt1 -> s1 := exp1;
  expBoolk |- evt1 -> s2 := exp2;
----------------------------------------
var guard_0, guard_1 : bool;
let
  guard_0 = false -> pre(expBooli) and evt1;
  guard_1 = false -> pre(expBoolj) and evt2;
  (s1,s2) = (s1Init,s2Init) -> if guard_0 then (exp1,pre(s2))
                               else if guard_1 then (pre(s1),exp2)
                               else (pre(s1),pre(s2)) // epsilon
tel
```

Plus simple, et traite correctement $\epsilon$.

# Les conditions initiales

Un événement n'est possible que si la situation au cycle précédent le permet. En lustre, il faut donc initialiser les variables (autres que les événements) pour ce premier cycle.

**flux in : public** : choisi par l'environnement.

**flux in : private** : fixé par un nœud.

**événement : public** : choisi par l'environnement. (actuellement la valeur du premier cycle est sans importance, car non prise en compte.

**événement : private** : fixé par un nœud. (non implémenté)

**état** fixé par la clause `init` sinon erreur possible.

**flux private & flux out** calculés par un nœud.

**garde :** sera évaluée, or elle dépend du cycle précédent, donc `true ->` pour la garde de $\epsilon$ et `false ->` pour toutes les autres.

# Que faire en cas de non-déterminisme ?

1. AltaRica est une description éventuellement non-déterministe.
2. Lustre est un programme déterministe.

Les choix :

1. Implantation déterministe d'une exécution possible.
2. Refuser toute description non-déterministe

Choix retenu : les deux (c'est une option).

# Traitement des événements

Pour chaque événement `ei` on associe :

1. une variable booléenne `ei`.
2. une variable booléenne `guard_j = false -> pre(exp) and ei`

Pour chaque ensemble d'événements `ei` d'un nœud, on associe :

1. une variable *virtuelle* `epsilon = not(OR(ei))`
2. une variable booléenne `guard_epsilon = epsilon`

puis des contraintes :

1. un et un seul événement `assert(#(e1,...,en))` (l'opérateur `#` dit au plus un, et la définition de $\epsilon$ assure l'unicité.)
2. au moins une transition possible `assert(OR(guard_i, guard_epsilon))`
3. au plus une transition possible `assert(#(guard_i, guard_epsilon))`

La dernière contrainte permet l'option.

# Traitement des assertions

**En AltaRica** les assertions sont des contraintes sur les configurations qui permettent soit de calculer les variables de flux, soit de refuser la transition.

```
assert fli = exp(fi1,...,fin,fl1,...,fln,s1,...,sn,n1.fioj,...,nn.fioj
       foi = exp(fi1,...,fin,fl1,...,fln,s1,...,sn,n1.fioj,...,nn.fioj
       expi(f1,...,fn,s1,...,sn,n1.fioj,...,nn.fioj);
```

**En Lustre** les assertions sont des contraintes sur les configurations. Les flux sont calculés par affectations.
La traduction est réalisée ainsi :

1. Pour les variables de flux `private` et `out`, la *première* assertion `f =` devient une affectation lustre.
2. Toutes les autres assertions deviennent des assertions lustre.

Tout source AltaRica *incorrect* produit un code refusé par Alta2Lustre.

# Et les post-conditions ?

**En AltaRica** si la post-condition est sans solution, c'est que la transition n'était pas possible. Ce point est surtout délicat en présence de priorité.

**En Lustre** les assertions jouent le même rôle, elles permettent de dire qu'un cycle de calcul n'est pas valide.
Il n'y a donc rien à faire de plus. Cela marche même dans les cas de synchronisations ou tous les sous-nœuds peuvent avancés, et c'est seulement au niveau principal qu'une assertion est violée.

# Bilan sur le langage

Un langage Alta2Lustre entre AltaRica et AltaRica DF pour lequel :

1. Un nouveau concept : l'événement local. (non traité)
2. Toute description déterministe est compilée en un programme lustre sémantiquement équivalent.
3. Pour toute description non déterministe, on génère au choix :
   - soit un programme lustre acceptant (choix en fonction de l'ordre des déclarations et $\epsilon$ en dernier). Ce choix combiné avec les *post-conditions* peut provoquer des *blocages* à l'exécution qui n'existent pas dans la description AltaRica, et qui n'existeraient pas avec d'autres choix déterministes.
   - soit un programme lustre non acceptant (erreur à l'exécution au moment du non-déterminisme).

# Bilan sur l'outil

Le compilateur Alta2Lustre produit soit :

1. des messages explicites pour certains sources AltaRica non valides.
2. un source lustre lisible et syntaxiquement incorrect, pour certains sources AltaRica non valides.
3. un source lustre syntaxiquement correct si le source est valide.
4. un source lustre dont les erreurs à l'exécution sont sémantiquement explicables.
5. sans doute quelques bugs ! (plusieurs niveaux de capture des événements)

Deux options actuellement disponibles :

**deterministe :** l'assertion qui garantit une et une seule transition est ajoutée.

**nomDuNoeud :** permet de compiler l'arborescence à partir de ce nœud. Sans l'option, le compilateur cherche un nœud `main`.

# Perspectives

**bugs** les corriger !

**événement local** doit pouvoir être traité syntaxiquement en utilisant la sémantique du pattern *englobant*, il faudra choisir une *priorité* entre les événements *locaux* et *globaux*.

**le type prédéfini integer** non implémenté actuellement dans les outils AltaRica, mais ne pose aucun problème pour le compilateur Alta2Lustre.

**les identificateurs** doivent pouvoir respecter la syntaxe AltaRica (caractère _ et le nom des événements identique à celui d'une variable).

**le formatage** des événements synchronisés doit être mieux réalisé.

**optimisation** les nœuds lustre sans variable locale n'ont pas à être dupliqués. Le nom du nœud AltaRica (et non l'instance) doit pouvoir être utilisé.

# Un dernier problème !

**A t'on bien choisi la bonne *sémantique* ?**