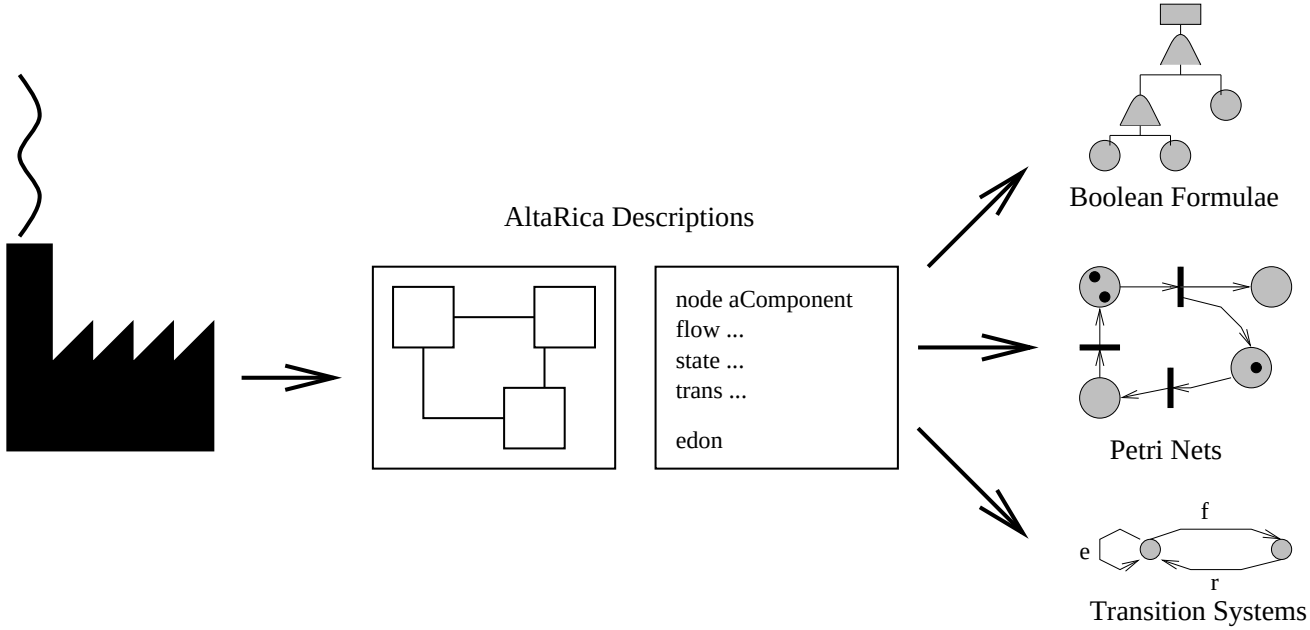
# ALTARICA : PASSÉ, PRÉSENT ET FUTUR

Alain Griffault & Antoine Rauzy

First AltaRica Workshop

Toulouse, 21 octobre 2002

# Le projet AltaRica




– Historique du projet.

**1997/2000** définition du langage et outillage.

**2000/...** atelier support et expérimentations.

– Particularités du projet.
  – Académique financé par des industriels.
  – Principalement dirigé par les besoins.

# Le modèle AltaRica

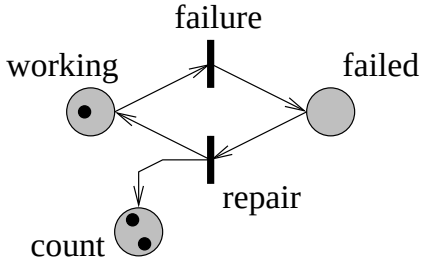
# Automates à contrainte

– un ensemble de variables $\vec{s}$,
– un ensemble d'événements $E$,
– un ensemble de transitions :

$$G(\vec{s}) \xrightarrow{e} \vec{s} := \sigma(\vec{s})$$

$G(\vec{s})$ est un(e) garde (c-à-d une expression booléenne) et $e \in E$,
– une assertion $A(\vec{s})$,
– des priorités (un ordre partiel) sur $E$.

Les automates à contraintes généralisent de nombreux modèles, par exemple les réseaux de Petri.




$\texttt{working} = 1 \xrightarrow{\texttt{failure}} \texttt{working} := 0, \texttt{failed} := 1$

$\texttt{failed} = 1 \xrightarrow{\texttt{repair}} \texttt{working} := 1, \texttt{failed} := 0, \texttt{count} := \texttt{count} + 1$

$\texttt{working} + \texttt{failed} = 1 \ \wedge \ \texttt{count} \leq 3$

# Le langage AltaRica

Le modèle n'est pas utilisable directement pour décrire un système complexe.
Les concepts introduits :

1. La hiérarchie.
2. Les interfaces (visibilité) des composants.
3. Description d'un noeud :
   - La distinction variable d'état/variable de flux.
   - Les transitions.
   - Les assertions : flux et états.
4. Description des interactions entre composants :
   - Les assertions : flux et états.
   - Synchrone : contrainte de synchronisation.
   - Asynchrone : flux et priorités.
5. Les directives outils.

# Syntaxe et sémantique AltaRica

# Syntaxe

```
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
```

Les rubriques sont toutes optionnelles.

# Sémantique : le temps AltaRica

1. Les événements sont asynchrones a priori.
2. Deux événements ne peuvent être simultanés que s'ils apparaissent dans une même contrainte de synchronisation.
3. Le système est événementiel.
4. Un événement n'est possible que si :
   - L'état du système satisfait sa garde.
   - Il existe au moins une valeur pour chacun des flux du système après les affectations mentionnées dans la transition.
   - L'événement est *prioritaire*.

# Sémantique : calcul

Deux solutions qui coïncident :

**produit synchronisé de produits synchronisés** en partant des feuilles.

**mise à plat** par réécriture puis sémantique du composant obtenu.

# Un premier exemple

```
// This generator could failed and be repaired.
node Generator
  flow   f1, f2 : bool;
  state  on : bool;
  event  failure, repair;
  trans  on |- failure -> on := false;
         not on |- repair -> on := true;
  assert on => (f1 and f2);
         not on => (not (f1 or f2));
//  extern initial_state = on = true;
edon
```

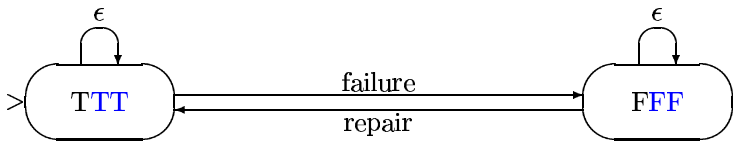



FIG. 1 – Un générateur (on,f1,f2)

# Un second exemple

```
node Switch
// No hypothesis about this switch.
  flow   f1, f2 : bool;
  state  on : bool;
  event  push;
  trans  true |- push -> on := not on;
  assert on => (f1 = f2);
edon
```

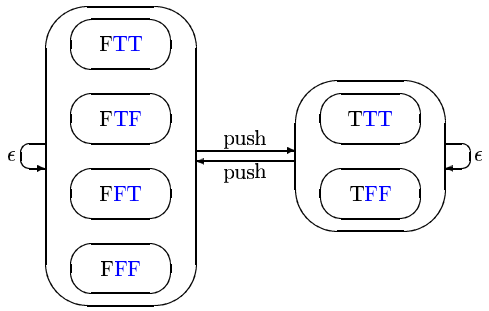



FIG. 2 – Un interrupteur (on,f1,f2)

**Remarque :** ne peut pas être ainsi décrit avec l'atelier OCAS, et en AltaRicaDF.

# Une première lampe

```
node Lamplight1
// This light is a lamp with a capacitor.
// So, there is a delay between:
//     the moment flows are changing,
//     and the moment the light is on or off.
  flow  f1, f2 : bool;
  state on : bool;
  event reaction;
  trans (f1 & f2) & (not on) |- reaction -> on := true;
        (not (f1 & f2)) & on |- reaction -> on := false;
//      (f1 & f2) != on |- reaction -> on := ~on;
edon
```

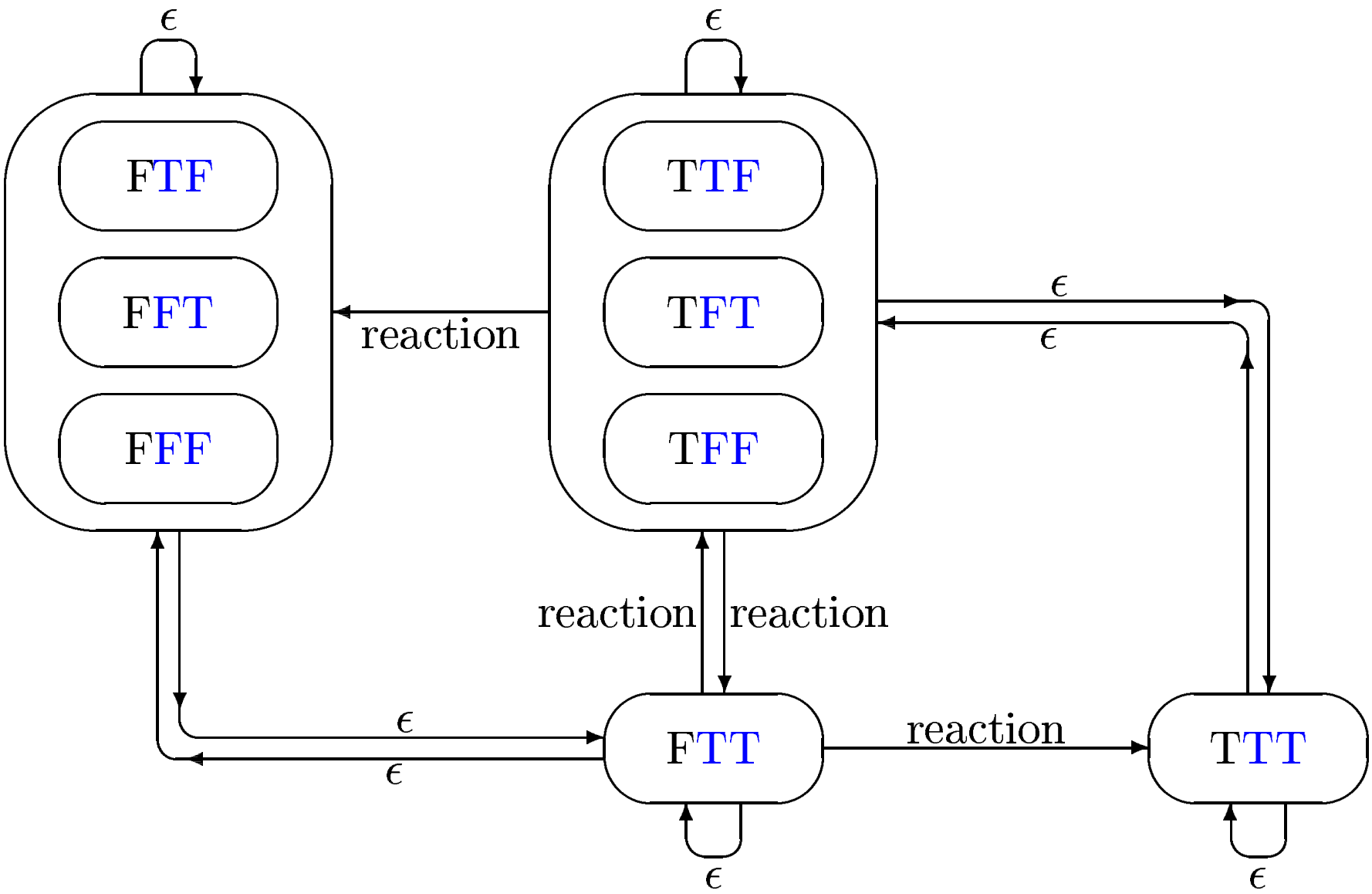



FIG. 3 – La lampe 1 (on,f1,f2)

# Un nœud AltaRica

**Les règles de visibilité**

| | visibilité | modification |
|---|---|---|
| état | | événement |
| flux | père | assert |
| événement | père | |

# Asynchronisme fort des composants

```
node main
  sub G : Generator;
      S : Switch;
      L : Lamplight1;
edon
```

- $2 \times 6 \times 8 = 96$ états.
- $2 \times 20 \times 32 + 2 \times 16 \times 32 + 2 \times 20 \times 16 + 2 \times 20 \times 32 = 4224$ transitions.

```
node main
  flow   G.f1, G.f2, S.f1, S.f2, L.f1, L.f2 : bool;
  state  G.on, S.on, L.on : bool;
  event  G.failure, G.repair, S.push, L.reaction;
  trans  G.on
             |- <eps,G.failure,S.eps,L.eps>  -> G.on := false;
         not G.on
             |- <eps,G.repair,S.eps,L.eps>   -> G.on := true;
         true
             |- <eps,G.eps,S.push,L.eps>     -> S.on := not S.on;
         (L.f1 & L.f2) & (not L.on)
             |- <eps,G.eps,S.eps,L.reaction> -> L.on := true;
         (not (L.f1 & L.f2)) & L.on
             |- <eps,G.eps,S.eps,L.reaction> -> L.on := false;
  assert G.on => (G.f1 and G.f2);
         not G.on => (not (G.f1 or G.f2));
         S.on => (S.f1 = S.f2);
//  extern initial_state = G.on = true;
edon
```

# Version 1 d'un premier système

```
node main
// The first global system.
  sub    G : Generator;
         S : Switch;
         L : Lamplight1;
// This part describes the connexions
//   between all the components.
  assert G.f1=S.f1;
         S.f2=L.f1;
         G.f2=L.f2;
edon
```

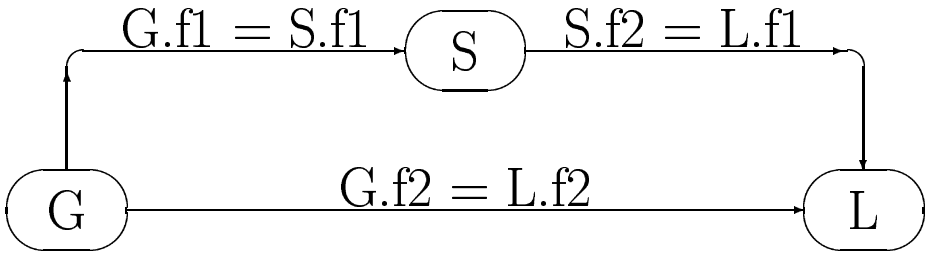



Figure 4: schéma d'un circuit

# Sémantique de la version A du premier système

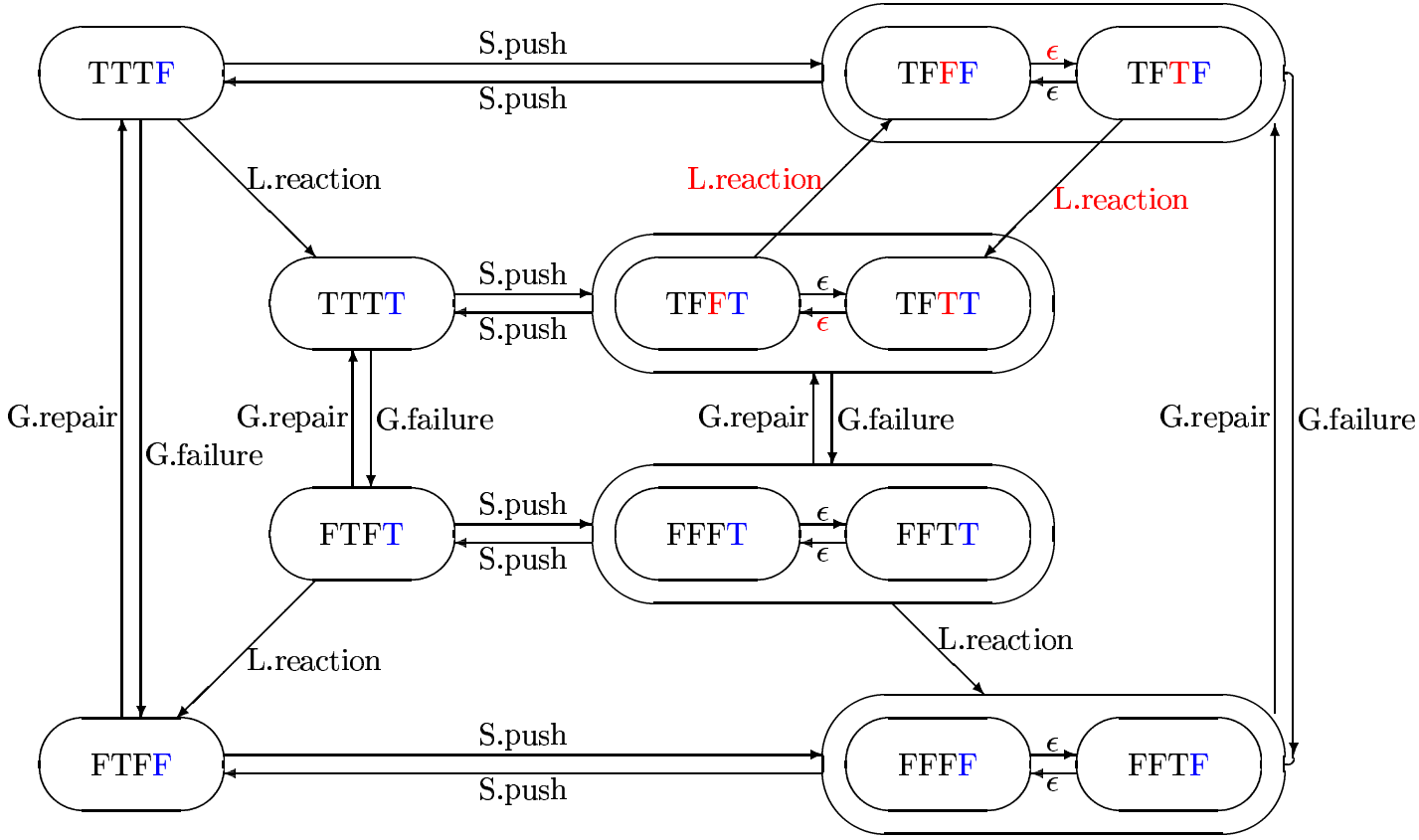



FIG. 5 – Le système 1.A (G.on,S.on,S.f2,L.on)

12 états et 66 transitions.

# L'ajout d'une contrainte

Afin d'enlever les réactions intempestives, il est nécessaire d'introduire la contrainte :
*Interrupteur(on=faux)* $\Rightarrow$ *Interrupteur(fluxDeSortie=faux)*
Deux méthodes :

**spécialisation d'un composant**

```
node OrientedSwitch
  flow   f1, f2 : bool;
  state  on : bool;
  event  push;
  trans  true |- push -> on := not on;
  assert on => (f1 = f2);
         f2 => on;
edon
```

**contrôle d'un composant**

```
node Switch
// No hypothesis about this switch.
// To observe the state of the switch,
//    we associate a flow to it.
  flow   f1, f2, stateOn : bool;
  state  on : bool;
  event  push;
  trans  true |- push -> on := not on;
  assert on => (f1 = f2);
         on = stateOn;
edon

node main
  assert
// To have a controled Switch in sense S.f2 = fct(S.f1,S.on)
//    we add this to direct the Switch
         S.f2 => S.stateOn;
edon
```

# Sémantique de la version B

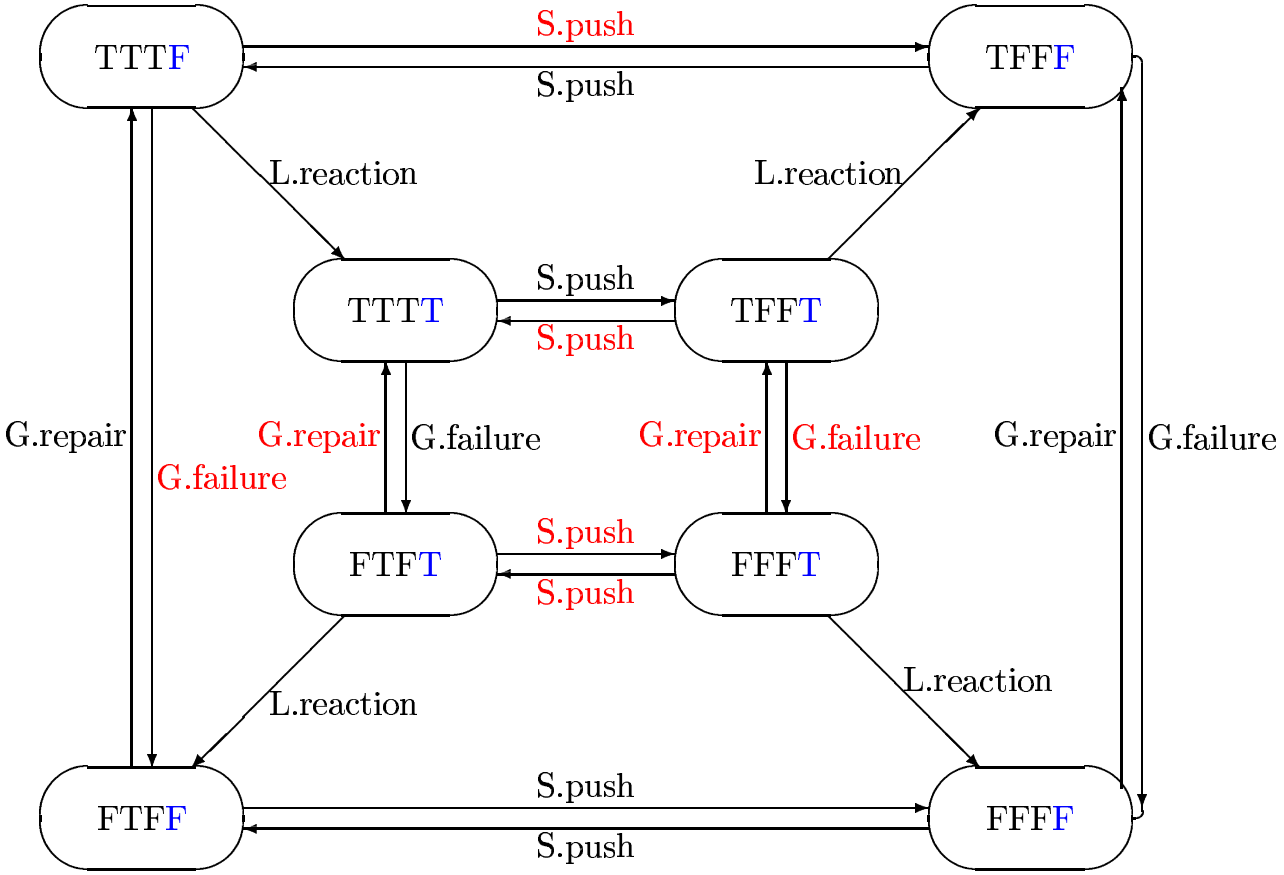



FIG. 6 – Le système 1.B (G.on,S.on,S.f2,L.on)

8 états et 28 transitions.

# Les priorités

```
node main
// two events to distinguish some priorities.
  event  a < {b};
  sub    G : Generator;
         S : Switch;
         L : Lamplight1;
  assert G.f1=S.f1;
         S.f2=L.f1;
         G.f2=L.f2;
         S.f2 => S.stateOn;  // To have an oriented Switch
  trans  true |- a,b ->;     // Not really transitions.
// The behaviors of the system.
  sync   <a,G.failure>;
         <a,G.repair>;
         <a,S.push>;
         <b,L.reaction>;
edon
```

# Sémantique de la version C

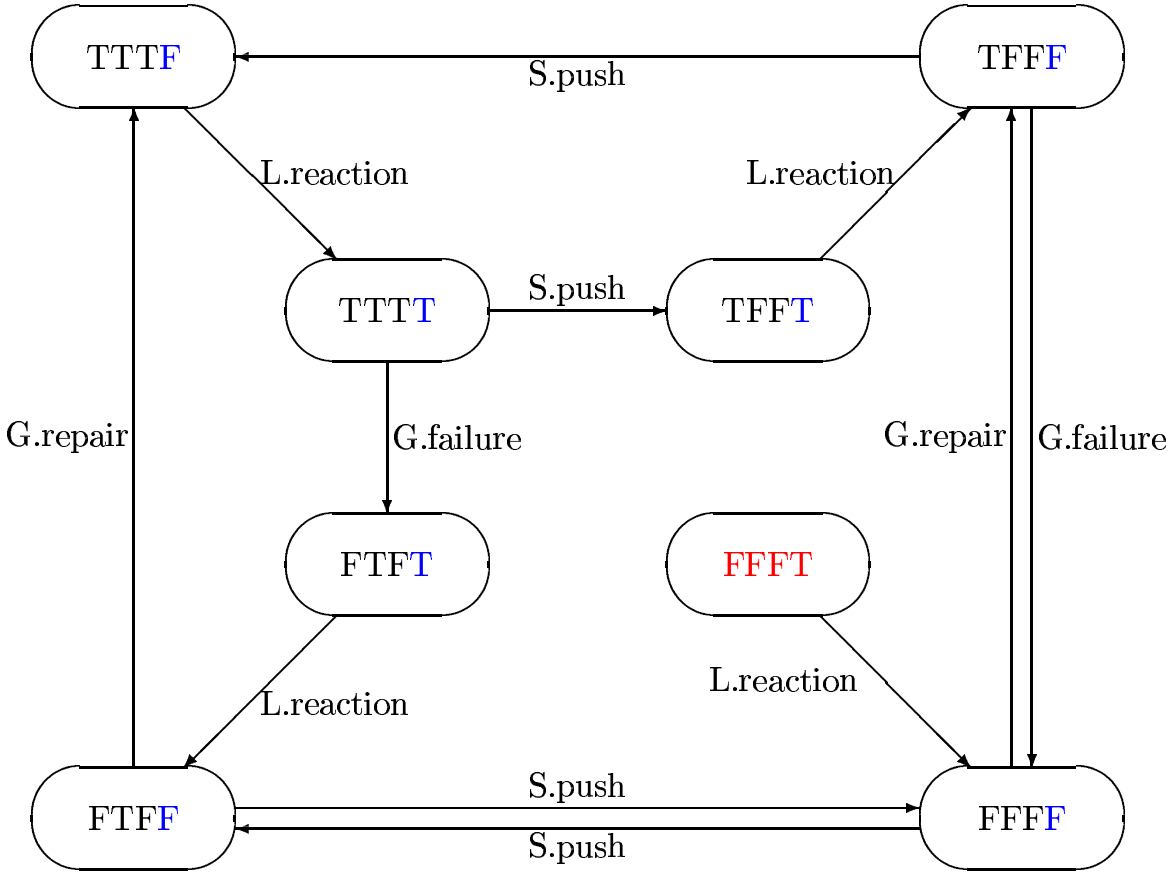



FIG. 7 – Le système 1.C (G.on,S.on,S.f2,L.on)

8 états et 20 transitions.

# Accessibilité et contraintes

**Ajout d'états initiaux et/ou d'une contrainte**

```
node main
...
  assert not(~G.on & ~S.on & ~S.f2 & L.on);
  extern initial_state = G.on = true, S.on = true;
edon
```

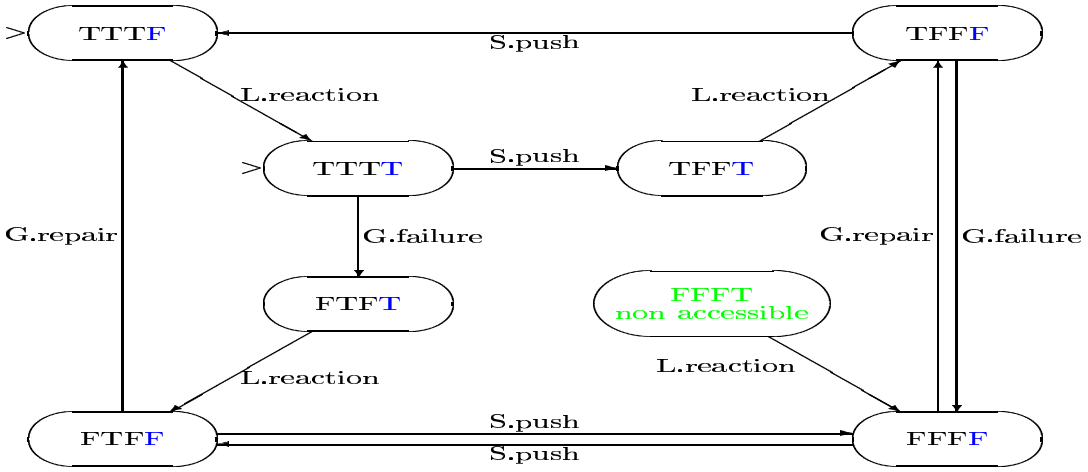



FIG. 8 – Le système 1.D (G.on,S.on,S.f2,L.on)

# Les contraintes de synchronisation

1. La synchronisation d'événements :

   **la synchronisation** pour dire que des événements ont lieu forcément en même temps,

   **la diffusion** pour dire que des événements, s'ils peuvent avoir lieu en même temps, alors seront simultanés.

   **une éventuelle cardinalité** pour indiquer des préférences.

2. La synchronisation d'états.

3. La synchronisation de propriétés de configuration.

# Une autre lampe

```
node Lamplight2
// This lamp reacts as soon as input flows change.
  flow   f1, f2 : bool;
  state  on : bool;
  event  reaction;
  trans  true |- reaction -> on := not on;
  assert on = (f1 & f2);
edon
```

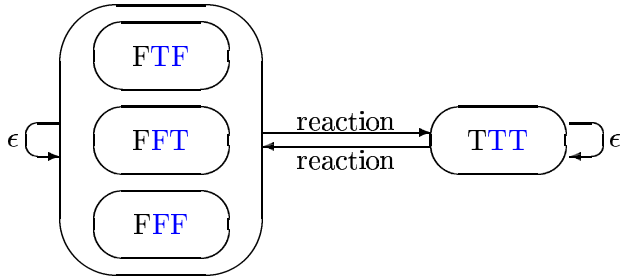



FIG. 9 – La lampe 3 (on,f1,f2)

# La diffusion

```
node main
  sub    G : Generator;
         S : Switch;
         L : Lamplight2;
  assert G.f1=S.f1; S.f2=L.f1; G.f2=L.f2;
         S.f2 => S.stateOn; // an oriented Switch
  sync   <G.failure, L.reaction?>;
         <G.repair, L.reaction?>;
         <S.push, L.reaction?>;
edon
```

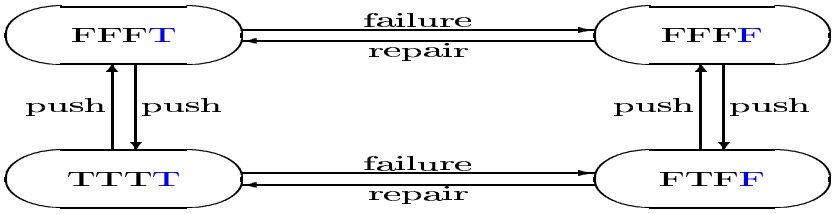



FIG. 10 – Le système 3.B (G.on,S.on,S.f2,L.On)

# Conception d'un modèle AltaRica

# Rappels des concepts

La description d'un modèle AltaRica utilise les concepts suivants :

1. La hiérarchie.
2. Les interfaces (visibilité) des composants.
3. Description d'un noeud :
   - La distinction variable d'état/variable de flux.
   - Les transitions.
   - Les assertions : flux et états.
4. Description des interactions entre composants :
   - Les assertions : flux et états.
   - Synchrone : contrainte de synchronisation.
   - Asynchrone : flux et priorités.
5. Les directives outils.

Il existe de nombreuses façons de décrire un système donné.

# Hiérarchie ou pas

La hiérarchie offre plusieurs avantages :

– une représentation plus "intuitive" soit de l'architecture, soit des fonctionnalités d'un système.
– une localisation accrue des interactions entre les composants.
– la possibilité de "cacher" certains composants.
– la possibilité de regrouper une "famille" de composants.
– l'étude des sous-systèmes.
– le remplacement aisé d'un sous-système par un autre ayant la même partie observable, et les mêmes événements.

Elle n'est pas adaptée lorsque la plupart des interactions mettent en jeu la majorité des composants.

# Qu'est ce qu'un événement

Un modèle AltaRica décrit un système événementiel, au sens où les seules évolutions perceptibles correspondent à un événement survenu dans tout ou partie du système.
La liste des événements est donc obtenue par l'union de tous les événements que vous souhaitez vouloir observer dans le système :

– les pannes des composants.
– les réparations des composants.
– les changements de modes des composants.
– les décisions locales aux composants.
– les fonctionnalités des composants.
– ...

# État/Événement ou flux

Une quantité mesurable d'un composant doit-elle être représentée par une variable d'état ou bien par une variable de flux ?

1. Cette quantité est-elle modifiable par l'environnement sans que le composant s'en aperçoive ?
2. Cette quantité est-elle modifiable seulement par le composant ?
3. Cette quantité est-elle une information dont l'environnement à besoin ?
4. Est-il nécessaire de mémoriser cette quantité pour des décisions ultérieures ?

| questions | réponses | état | flux |
|---|---|---|---|
| 1 | oui | | X |
| 2 | oui | X | |
| 3 | oui | | X |
| 4 | oui | X | |

# Quelques modèles de description

1. L'exportation d'une variable d'état pour la visualisation et le contrôle.
2. La propagation de l'interface (flux et événements) d'un composant.
3. L'adaptateur d'un nœud pour la réutilisation.
4. L'agrégation de plusieurs événements `sync <e,a1.e?,a2.e?>=1;`.
5. La causalité entre événements pour propager une information.
6. Le contrôle de sous-systèmes avec les priorités.
7. Le contrôle de sous-systèmes avec les flux.
8. Les variables globales pour la modification et la consultation d'une entité par un ensemble de composants.
9. La transmission sûre d'une valeur permet une abstraction d'un protocole fiable entre deux composants.
10. L'exclusion mutuelle avec la synchronisation par diffusion.
11. L'exclusion mutuelle avec les flux.
12. Les abstractions : le non-déterminisme des flux et des états

Un bilan des trois dernières années

# Les outils disponibles en janvier 2000

1. Les briques :

   **le parser** lit un source et produit l'arbre syntaxique.

   **la mise à plat** produit un modèle AltaRica sémantiquement correct.

   **le solveur** permet de calculer les transitions possibles à partir d'une configuration.

2. Les outils :

   **le simulateur** permet d'exécuter des scénarios.

   **a-aralia** génère un arbre de défaillance.

   **a-mec** génère un système de transition.

   **un AGL** permet la saisie d'un modèle.

# Les cas d'utilisation

**Les benchmarks et les tests** pour valider le langage et les outils.

**La sûreté de fonctionnement** utilise le simulateur, le générateur d'arbres de défaillance et la simulation stochastique,

**La conception de systèmes distribués** utilise le simulateur et le générateur de systèmes de transition.

# Les limites et/ou défauts constatés

Le langage est bien conçu et bien adapté mais :

**Les benchmarks et les tests** montrent :
- la diffusion (la priorité) coûte cher en CPU.

**La sûreté de fonctionnement** montre :
- le simulateur est quelque fois trop lent (diffusion),
- le générateur d'arbre de défaillance est assez inefficace,
- l'AGL est mal conçu,
- quelques *défauts* dans le langage.

**La conception de systèmes distribués** montre :
- l'exploitation du système de transition obtenu est difficile.
- trop souvent, le langage est **WYGINWYW**.

**Les bugs** existent mais semblent peu nombreux.

Les patchs réalisés

# Les développements Dassault/IXI

Dassault Aviation et IXI développent l'AGL OCAS orienté métier qui :

– limite le langage (pas de priorité et pas de synchronisation),

– ajoute des fonctionnalités *pratiques* :

  le flux `icône` hors de la rubrique externe pour l'animation,

  la possibilité de définir des structures,

  l'événement `no_event` qui est *ultra prioritaire* pour traiter la propagation,

  le mode `equipment` qui permet de remonter les événements automatiquement, pour les pannes notamment,

  les attributs `in, out` pour orienter les flux.

– remplace le solveur par un ATMS,

– réécrit le générateur d'arbres de défaillance,

– contient de nombreuses bibliothèques.

# Les développements ARboost technologies

Antoine définit le langage AltaRicaDF et propose pour ce langage un générateur d'arbres de défaillance et un simulateur stochastique.
AltaRicaDF limite et modifie AltaRica

1. Les modifications :
   - les flux sont orientés : `in|out|local`
   - les flux et les états sont visibles par la hiérarchie (boites blanches),
   - les priorités sont entières (ordre total),
   - une clause `init`.
2. Les limites :
   - la diffusion n'est pas permise,
   - il n'y a pas de comportement dans les nœuds,
   - les assertions sont limitées à la définition des flux de sorties,
   - un nœud `main` obligatoire.

# Les développements LaBRI

Avec le CERT-ONERA dans le cadre d'ESACS, définition du langage AltaRicaESACS, et développement des compilateurs :

- AltaRicaESACS → lustre
- AltaRicaESACS → AltaRica
- lustre → AltaRicaESACS

AltaRicaESACS limite et modifie AltaRica :

1. Les modifications :
   - un attribut de visibilité `public, parent, private` est attaché aux flux, états et événements (boite avec une partie noire et une partie blanche)
   - les flux non `private` sont orientés : `in|out`
2. Les limites :
   - la diffusion et les priorités ne sont pas permises,
   - les assertions doivent définir les flux de sorties.

Afin de rendre le langage moins **WYGINWYW** et de remplacer `a-mec` :

**acheck** pour vérifier des comportements sur des petits modèles AltaRica.

**mec** (en cours) pour le faire sur des (moins) petits modèles.

## Les développements IRCCyN

Olivier Roux, Franck Cassez et Claire Pagetti travaillent sur une extension temporisée du modèle AltaRica.

$\Rightarrow$ ce soir ici même à 15 heures 45.

# Les développements CERT-ONERA

1. Une amélioration du simulateur Java de Gérald qui permet :
   - de visualiser la hiérarchie,
   - découplage client/serveur.
2. De nombreuses études de cas en collaboration avec l'Aérospatiale.
3. Dans le cadre du projet européen ESACS :
   - promotion du langage AltaRica et de l'AGL OCAS,
   - constitution de *pattern de fiabilité*,
   - utilisation des différents outils AltaRica.

$\Rightarrow$ Il faut harmoniser les dialectes et faire se rencontrer les utilisateurs...

Merci CHRISTEL

# Proposition d'extension du langage AltaRica

# Pourquoi faut il modifier AltaRica ?

Les objectifs de l'harmonisation :

1. l'échange des modèles (syntaxe compréhensible par tous les outils),
2. la véracité des faits (même interprétation sémantique d'un source),
3. éviter les outils *passerelle*.

Le seul moyen fiable pour y arriver :

1. un langage *englobant* et une sémantique $\rightarrow$ *les mêmes briques*.
2. des sous-langages qui ne sont que des restrictions syntaxiques $\rightarrow$ *des outils différents*.

# Comment faut il modifier AltaRica ?

**Les besoins et leurs conséquences sur les briques les outils**

| | | | |
|---|---|---|---|
| | AltaRicaDF | | |
| flux | `in, out, local` | Non | Oui |
| flux et états | visibles dans la hiérarchie | Non | Oui |
| événements | *remontés* jusqu'au `main` | **Oui** | Oui |
| priorité | entière | **Oui** | Oui |
| | clause init | Non | Oui |
| | OCAS | | |
| flux | icône | Non | Oui |
| flux | `in, out` | Non | Oui |
| | les structures | Non | Oui |
| événements | `no_event` | **Oui** | Oui |
| extern | mode `equipment` | **Oui** | Oui |
| | AltaRicaESACS | | |
| flux | `in, out` | Non | Oui |
| visibilité | `private, parent, public` | **Oui** | Oui |

# Une proposition pour AltaRica

Une liste d'attributs attachée à chaque flux, état et événement.

| | **visibilité** | **autres** | **utilisations** |
|---|---|---|---|
| `flow` | `private` | ... | *variables locales* |
| | `parent` | `in, out` ... | AltaRicaESACS pour *arrêter des flux* |
| | `public` | `in, out` ... | défaut d'AltaRicaDF et pratique pour OCAS |
| `state` | `private` | ... | ancienne sémantique et AltaRicaESACS |
| | `parent` | ... | pour l'observation et le contrôle |
| | `public` | ... | défaut d'AltaRicaDF et pour l'observation |
| `event` | `private` | ... | permet de traiter `no_event` d'OCAS |
| | `parent` | ... | ancienne sémantique et AltaRicaESACS |
| | `public` | ... | défaut d'AltaRicaDF et pour les pannes |

La clause `init` est possible.

Les quatre langages AltaRicaOld, AltaRicaOCAS, AltaRicaDF et AltaRicaESACS sont tous des sous-langages définissables par des restrictions syntaxiques.

# Les questions et/ou choix non résolus

1. Les priorités entières et définies par ordre partiel sont-elles facilement compatibles ?
2. Quelle est la sémantique en cas d'absence d'un attribut `private, parent` ou `public` pour les événements ?