Conception Formelles

Alain Griffault
Université Bordeaux I

Grégoire Sutre
CNRS

28 octobre 2004

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Les techniques formelles : pour qui ? pour quoi ?

L'ingénierie des systèmes intègre de plus en plus dans son cycle de vie des moyens rigoureux (basés sur des définitions mathématiques) dont l'utilisation doit augmenter, par une meilleure connaissance d'un modèle, la confiance qu'un utilisateur à dans une conception et/ou une réalisation d'un système.

Pour des raisons économiques, le surcroît de travail fait que seuls les concepteurs et/ou réalisateurs de systèmes complexes et/ou critiques s'intéressent aujourd'hui aux techniques formelles. Un système est généralement considéré comme tel lorsque :

- la vie de personnes est lié à son bon fonctionnement. C'est le cas des logiciels embarqués pour le transport (avion, train, voiture, navette...), des contrôleurs de systèmes (centrales nucléaires, matériels médicaux...).
- le coût économique d'un dysfonctionnement est catastrophique. C'est le cas des logiciels mis dans le silicium pour être produit en très grande quantité (électroménager, téléphonie...)

Les techniques formelles utilisées en ingénierie des systèmes sont nombreuses et diverses.

1.2 Un rapide panorama des techniques formelles

Les démonstrateurs de théorèmes sont des outils qui aident à fabriquer des preuves **justes**. Un programme est vu comme une fonction de calcul, et on va chercher à montrer que d'une part la fonction termine ; d'autre part que le résultat calculé est bien celui attendu.

Cette technique est bien adaptée pour des programmes séquentiels complexes, mais moins bien pour les systèmes de types réactifs.

L'interprétation abstraite consiste à transformer un problème concret dans un modèle abstrait plus simple, sur lequel les *preuves* seront plus faciles à effectuer. Un exemple d'interprétation abstraite est la preuve par 9.

Cette technique est bien adaptée pour les programmes manipulant beaucoup de données, mais moins bien pour les systèmes de types réactifs.

Le raffinement permet, en un nombre d'étapes plus ou moins important de passer d'une spécification à une implantation en préservant à chaque étape les propriétés essentielles du système.

Cette technique est bien adaptée pour des programmes séquentiels complexes, mais moins bien pour les systèmes de types réactifs.

La génération automatique de codes exécutables transforme une description, dans un langage de très haut niveau et très spécialisé, en un programme dans un langage *classique* (C ou ADA).

Cette technique est bien adaptée pour les systèmes temps réels.

Les vérificateurs de modèles sont des outils qui permettent de décider si un modèle d'un système satisfait ou non des propriétés logiques. Dans cette démarche, le modèle est une représentation du système, et les propriétés représentent les fonctionnalités attendues du système.

Cette technique est très bien adaptée pour la conception des systèmes réactifs, mais manipule assez mal les données. En pratique, les utilisateurs appliquent *manuellement* des techniques d'interprétation abstraite lorsque le système possèdent de trop nombreuses données.

Les générateurs de tests sont des outils qui permettent de générer des séquences de tests qui ont la particularité d'être d'une part conforme aux spécifications ; d'autre part conforme à des objectifs de tests spécifiques. Ils permettent ainsi de contrôler que le système réel est plus ou moins conforme avec le cahier des charges initial.

Cette technique est très bien adaptée pour les systèmes répartis et notamment pour les protocoles. Elle se démarque des précédentes du fait que c'est la seule à faire intervenir le système réel.

Les processus stochastiques ajoutent dans la description des systèmes des grandeurs probabilistes. Il est alors possible de calculer des statistiques relatives à des propriétés. La sûreté de fonctionnement qui cherche à maîtriser les systèmes en présence de dysfonctionnements est un des domaines applicatifs des processus stochastiques.

1.3 Techniques formelles et cycle de développement

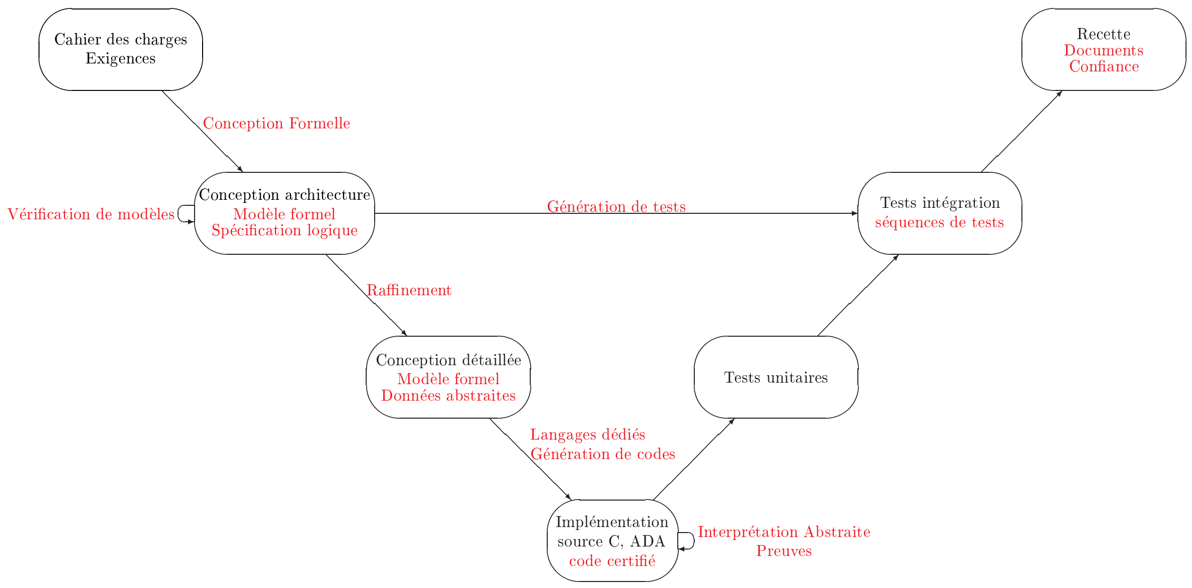


FIG. 1.1 – Un classique cycle en V décoré de techniques formelles

1.4 Contenu du cours

1. Conceptions formelles et vérification de modèles avec le langage ALTARICAet les outils `altatools` et `MecV`.
2. Raffinement et génération de codes avec le langage B et les outils `Atelier B` et `B4free`.

Première partie

Le langage AltaRica

Chapitre 2

Historique du projet AltaRica

2.1 Le projet AltaRica

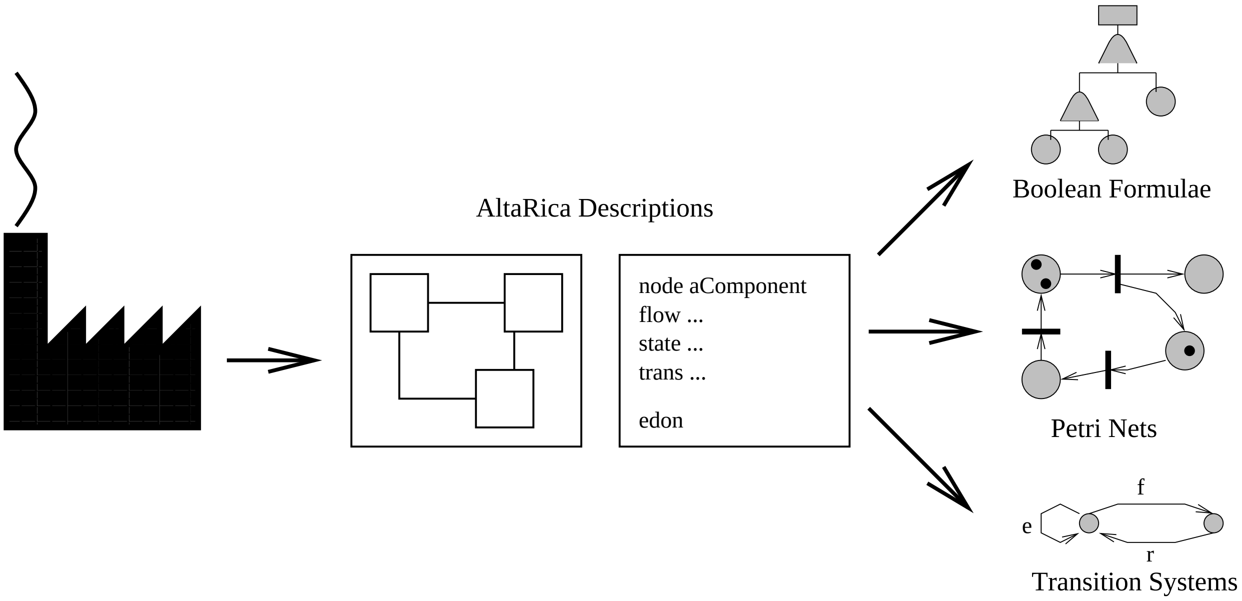


Fig. 2.1 – Le cahier des charges pour AltaRica

- 97/99 : Définition du langage et premier prototype.
- **Académiques :** LaBRI and LADS et **partenaires industriels :** Dassault, Thales, Total, Schneider, Renault, IPSN, CEA, GFI
- Deux requêtes principales :
  1. Le langage doit être adapté à l'utilisateur.
  2. Une description unique pour garantir une cohérence.

2.2 Les besoins industriels

- *rigueur* (i.e pas un langage semi-formel),
- à la fois graphique et textuel,
- non spécifique à un domaine (avionique, nucléaire, électricité, ...),

- hiérarchique,
- à la fois pour des analyses fonctionnelles et dysfonctionnelles,
- tous les types de communication,
- facile d'utilisation,
- aucun besoin de génération de codes (doit être utilisé en conception de systèmes)
- ...

2.3 L'état actuel du projet AltaRica

- Trois boites à outils :

  OCAS AltaRica Dassault (simulation, arbre de défaillance),

  AltaRicaDF A. Rauzy (stochastique, arbre de défaillance),

  altatools LaBRI (sémantique et vérification de modèles).
- Des projets communs :

  T_AltaRica (IRCCyN)

  AltaRica↔`Lustre` (ONERA).
- deux workshops (10/2002 and 10/2003).
- L'atelier OCAS est en cours de certification par les instances avioniques européennes et américaines.

Chapitre 3

Le langage AltaRica

3.1 Le modèle de calcul

Definition 3.1 *Un automates à contraintes est un triplet :*
- *un ensemble de variables $\vec{s}$,*
- *un ensemble d'événements E,*
- *un ensemble de transitions :*

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

$G(\vec{s})$ est un(e) garde (c-à-d une expression booléenne) et $e \in E$,

3.2 Le premier exemple

Un interrupteur est ouvert ou fermé.

```
node  Interrupteur
state ouvert : bool;
event push;
trans ouvert = true  |- push -> ouvert := false;
      ouvert = false |- push -> ouvert := true;
edon
```

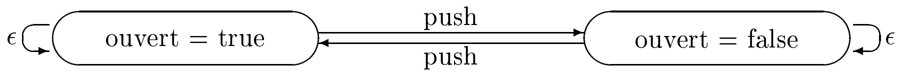


Fig. 3.1 – L'automate sémantique du premier exemple

Une version plus élégante du même composant :

```
node  Interrupteur
state ouvert : bool;
event push;
trans true |- push -> ouvert := not ouvert;
edon
```

3.3 Un premier problème

Trois descriptions équivalentes d'un buffer de taille 2,

```
node  FIFO_V1
state nbMessages : [0,2];
event put, get;
trans nbMessages < 2 |- put -> nbMessages := nbMessages + 1;
      nbMessages > 0 |- get -> nbMessages := nbMessages - 1;
init  nbMessages := 0;
edon
```

```
node  FIFO_V2
state nbMessages : int;
event put, get;
trans nbMessages >= 0 & nbMessages < 2 |- put -> nbMessages := nbMessages + 1;
      nbMessages <= 2 & nbMessages > 0 |- get -> nbMessages := nbMessages - 1;
init  nbMessages := 0;
edon
```

```
node  FIFO_V3
state nbMessages : [0,2];
event put, get;
trans true |- put -> nbMessages := nbMessages + 1;
      true |- get -> nbMessages := nbMessages - 1;
init  nbMessages := 0;
edon
```

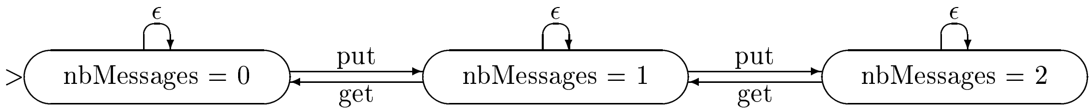


FIG. 3.2 – L'automate sémantique des trois versions de FIFO

En ALTARICA, l'affectation n'a lieu que si elle est possible. Le modèle est dit à post-condition.

Question 3.1 *Écrivez le code* ALTARICA *d'un processus de l'algorithme d'exclusion mutuelle de Peterson dans le cas de deux processus.*

```
while true do begin
              F[i] := true;
              Tour := i;
              wait (F[j]=false or Tour = j);
              F[i] := false;
              end;
```

Corrigé question 3.1

```
node ProcPeterson
state co :[0,3];
      Tour,i,j : [0,1];
      Fi, Fj : bool;
event run;
```

```
trans co = 0              |- run -> co := 1, Fi := true;
      co = 1              |- run -> co := 2, Tour := i;
      co = 2 & ~Fj        |- run -> co := 3;
      co = 2 & (Tour = j) |- run -> co := 3;
      co = 3              |- run -> co := 0, Fi := true;
init  Fi := false, Fi := false;
edon
```

3.4 Un deuxième exemple

Une trajectoire :

```
node   Trajectoire
state  x : [-5,5];
flow   y : int;
event  droite, gauche;
trans  true  |- droite -> x := x+1;
       true  |- gauche -> x := x-1;
assert y = (x*x -4*x +3);
edon
```

L'affectation est parallèle, le calcul de y est difficile en terme d'affectations.

L'assertion est une formule booléenne qui restreint les valuations possibles des variables.

En AltaRica, deux types de variables : les variables d'états qui sont affectées, et les variables de flux qui sont contraintes.

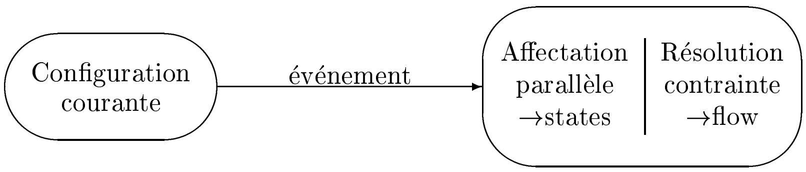


Fig. 3.3 – Sémantique franchissement d'une transition

Question 3.2 *Écrivez le code* AltaRica *d'un interrupteur indiquant de plus la présence d'une tension à ses deux extrémités.*

Corrigé question 3.2

```
node   Interrupteur
state  ouvert : bool;
flow   f1, f2 : [0,1];
event  push;
trans  true |- push -> ouvert := ~ouvert;
assert ~ouvert => (f1 = f2);
edon
```

3.5 Un premier exemple hiérarchique

Un circuit électrique est composé d'un générateur pouvant tomber en panne et être réparé, d'un interrupteur et d'une ampoule.

```
node   Generateur
state  ok : bool;
flow   plus, neutre : [0,1];
event  panneCC, reparation;
trans  ok  |- panneCC    -> ok := false;
       ~ok |- reparation -> ok := true;
assert plus = 1;
       ok = (neutre = 0);
init   ok := true;
edon
```

Question 3.3 *Dessinez le système de transition correspondant.*

Corrigé question 3.3

```
node   Interrupteur
state  ouvert : bool;
flow   f1, f2 : [0,1];
event  push;
trans  true |- push -> ouvert := ~ouvert;
assert ~ouvert => (f1 = f2);
edon
```

Question 3.4 *Dessinez le système de transition correspondant.*

Corrigé question 3.4

```
node   Lampe_V1
state  allume, grille : bool;
flow   f1, f2 : [0,1];
event  reaction;
trans  ~grille & (f1 = 1 & f2 = 1) |- reaction -> grille := true, allume := false;
       ~grille & (allume = (f1 = f2) |- reaction -> allume := ~allume;
init   grille := false;
edon
```

Question 3.5 *Dessinez le système de transition correspondant.*

Corrigé question 3.5

```
node   Circuit_V1
sub    G : Generateur;
       I : Interrupteur;
       L : Lampe_V1;
assert G.plus = I.f1;
       I.f2 = L.f1;
       G.neutre = L.f2;
edon
```

Par défaut, les événements des sous-composants sont asynchrones.
La description précédente est équivalente à la suivante :

```
node   Circuit_V1
state  G.ok : bool;
       I.ouvert : bool;
       L.allume, L.grille : bool;
flow   G.plus, G.neutre : [0,1];
       I.f1, I.f2 : [0,1];
       L.f1, L.f2 : [0,1];
event  <G.panneCC>, <G.reparation>, <I.push>, <L.reaction>;
trans  G.ok |- <G.panneCC> -> G.ok := false;
       ~G.ok |- <G.reparation> -> G.ok := true;
       true |- <I.push> -> I.ouvert := ~I.ouvert;
       ~L.grille & (L.f1 = 1 & L.f2 = 1) |- <L.reaction> -> L.grille := true, L.allume := false;
       ~L.grille & (L.allume = (L.f1 = L.f2) |- <L.reaction> -> L.allume := ~L.allume;
assert G.plus = 1;
       G.ok = (G.neutre = 0);
       ~I.ouvert => (I.f1 = I.f2);
       G.plus = I.f1;
       I.f2 = L.f1;
       G.neutre = L.f2;
init   G.ok := true, L.grille := false;
edon
```

Il est possible de préciser que des événements sont synchrones. On peut par exemple ajouter une réparation à la `Lampe`, et imposer que les réparations du `Générateur` et de la `Lampe` soient simultanés.

```
node   Lampe_V2
state  allume, grille : bool;
flow   f1, f2 : [0,1];
event  reaction, reparation;
trans  ~grille & (f1 = 1 & f2 = 1) |- reaction -> grille := true, allume := false;
       ~grille & (allume = (f1 = f2) |- reaction -> allume := ~allume;
       grille |- reparation -> grille := false;
init   grille := false;
edon

node   Circuit_V2
sub    G : Generateur;
       I : Interrupteur;
       L : Lampe_V2;
sync   <G.reparation, L.reparation>;
assert G.plus = I.f1;
       I.f2 = L.f1;
       G.neutre = L.f2;
edon
```

Cela modifie ainsi le comportement global

```
node   Circuit
state  ...
flow   ...
event  <G.panneCC>, <G.reparation, L.reparation>, <I.push>, <L.reaction>;
trans  ~G.ok & L.grille |- <G.reparation, L.reparation> -> G.ok := true, L.grille := false;
```

```
       ...
assert ...
init   ...
edon
```

La construction hiérarchique n'est pas limitée. De plus, un nœud père est lui-même un automate à contrainte ; c'est à dire qu'il peut avoir ses propres variables, événements et transitions. Par exemple, afin d'assurer que les réparations se font en situation de sécurité, on peut ajouter :

```
node   Circuit_V3
state  sur : bool;
event  debutIntervention, finIntervention, reparation;
trans  sur  |- reparation -> ;
       sur  |- finIntervention -> sur := false;
       ~sur |- debutIntervention -> sur := true;
sub    G : Generateur;
       I : Interrupteur;
       L : Lampe_V2;
sync   <reparation, G.reparation, L.reparation>;
assert ~sur => (G.plus = I.f1);
       sur => (I.f1 = 0);
       I.f2 = L.f1;
       ~sur => (G.neutre = L.f2);
       sur => (L.f2 = 0);
init   sur := false;
edon
```

Question 3.6 *Écrivez le code* ALTARICA *de l'algorithme de Peterson dans le cas de deux processus.*

Corrigé question 3.6

```
node ProcPeterson
  state
    co : [0,3];
    fi : bool;
  flow
    fj : bool;
    tour, i : [0,1];
  event
    run, TourToI ;
  trans
    co=0                      |- run     -> co := 1, fi := true;
    co=1                      |- TourToI -> co := 2;
    co=2 & (~fj | (tour != i) |- run     -> co := 3;
    co=3                      |- run     -> co := 0, fi := false;
  init
    co := 0, fi := false;
edon

node Peterson2
  sub
    P0, P1 : Processus;
  state
    tour : [0,1];
```

```
  event
    P0assignTour, P1assignTour;
  trans
    true |- P0assignTour -> tour := P0.i;
    true |- P1assignTour -> tour := P1.i;
  sync
    <P0assignTour, P0.TourToI>;
    <P1assignTour, P1.TourToI>;
  assert
    P0.i = 0;
    P1.i = 1;
    P0.tour = tour;
    P1.tour = tour;
    P0.fi = P1.fj;
    P1.fi = P0.fj;
edon
```

Question 3.7 *Essayez de trouver des* erreurs *dans le modèle du* `Circuit`*.*

Corrigé question 3.7

3.6 Le concept de priorité

3.6.1 Un problème

Considérons un ordonnanceur basé sur des files d'attentes.

```
node PoolOfJobs
  state nb : [0,2];
  event put, get;
  trans nb<2 |- put -> nb := nb+1;
        nb>0 |- get -> nb := nb-1;
  init  nb := 0;
edon

node SchedulerRandom
  sub P0, P1, P2 : PoolOfJobs;
edon
```

Comme son nom l'indique, cet ordonnanceur choisit aléatoirement la tâche a exécutée dans une quelconque des files non vides.

Question 3.8 *Écrivez le code* AltaRica *d'un ordonnanceur qui prend une tâche dans la nième file seulement si toutes les files d'indice inférieur sont vides..*

Corrigé question 3.8
Pour modéliser des priorités entre les files, une solution consiste à modifier le modèle ainsi.

```
node PoolOfJobs
  state nb : [0,2] : parent;
  event put, get;
  trans nb<2 |- put -> nb := nb+1;
        nb>0 |- get -> nb := nb-1;
  init  nb := 0;
```

```
edon

node SchedulerPriorite
  sub P0, P1, P2 : PoolOfJobs;
  event run0, run1, run2;
  trans true              |- run0 -> ;
        P0.nb=0           |- run1 -> ;
        P0.nb=0 & P1.nb=0 |- run2 -> ;
  sync  <run0, P0.get>;
        <run1, P1.get>;
        <run2, P2.get>;
edon
```

Il faut pour chaque événement, ajouter dans sa garde, la conjonction des négations des gardes des événements qui lui sont prioritaires.

3.6.2 Objectifs

La modélisation des priorités peut s'avérer difficile. Le concept de priorité AltaRica peut-être vu comme une *macro* remplaçant cette écriture par une déclaration de priorité entre les événements.

3.6.3 Un exemple

Le modèle suivant utilisant les priorités répond également à la question 3.8.

```
node PoolOfJobs
  state nb : [0,2];
  event put, get;
  trans nb<2 |- put -> nb := nb+1;
        nb>0 |- get -> nb := nb-1;
  init  nb := 0;
edon

node Scheduler
  sub P0, P1, P2 : PoolOfJobs;
  event run0, run1, run2;
        run0 > run1; run1 > run2;
  trans true |- run0, run1, run2 -> ;
  sync  <run0, P0.get>;
        <run1, P1.get>;
        <run2, P2.get>;
edon
```

3.6.4 Idée intuitive de la sémantique

Les priorités sont définies par un ordre partiel sur les événements locaux. La sémantique fait qu'un événement n'est possible, que si tous les événements plus prioritaires ne le sont pas.

3.7 Le concept de diffusion

3.7.1 Un problème

Considérons un enseignant qui parle et distribue des polycopiés à ses élèves. On veut décrire le fait que les étudiants écoutent l'enseignant et écrivent sur les polycopiés.

```
node Eleve
  state present, crayon : bool;
  event in_out, ecoute, ecrit;
  trans true |- in_out -> present := ~present, crayon := true;
        true |- in_out -> present := ~present, crayon := false;
        present |- ecoute -> ;
        present & crayon |- ecrit -> ;
edon

node Enseignant
  state present : bool;
  event in_out, parle, polycop;
  trans true |- in_out -> present := ~present;
        present |- parle, polycop -> ;
edon

node CoursDifficile
  sub   En : Enseignant;
        El0, El1, El2 : Eleve;
  sync  <En.parle, El0.ecoute, El1.ecoute, El2.ecoute>;
        <En.polycop, El0.ecrit, El1.ecrit, El2.ecrit>;
edon
```

Le cours est difficile, car du fait de la sémantique des synchronisations, un enseignant ne peut parler que si **tous** les élèves sont présents ; et ne peut distribuer son polycopié que si **tous** les élèves sont présents et possèdent un crayon.

Question 3.9 *Écrivez un code* ALTARICA *qui :*

1. *quel que soit le nombre d'élèves présents, permet à l'enseignant de faire normalement son cours,*
2. *quel que soit le nombre d'élèves présents et ayant un crayon, permet à l'enseignant de distribuer au maximum ses deux exemplaires du polycopié.*

Corrigé question 3.9
En utilisant le mécanisme de priorité, la solution suivante est correcte.

```
node Eleve
  state present, crayon : bool;
  event in_out, ecoute, ecrit;
  trans true |- in_out -> present := ~present, crayon := true;
        true |- in_out -> present := ~present, crayon := false;
        present |- ecoute -> ;
        present & crayon |- ecrit -> ;
edon

node Enseignant
  state present : bool;
  event in_out, parle, polycop;
  trans true |- in_out -> present := ~present;
        present |- parle, polycop -> ;
edon

node CoursNormal
  sub   En : Enseignant;
        El0, El1, El2 : Eleve;
```

```
  event ec0 < ec1 < ec2 < ec3;
        p0 < p1 < p2;
  trans true |- ec0, ec1, ec2, ec3, p0, p1, p2 -> ;
  sync  <ec3, En.parle, El0.ecoute, El1.ecoute, El2.ecoute>;
        <ec2, En.parle, El0.ecoute, El1.ecoute>;
        <ec2, En.parle, El0.ecoute, El2.ecoute>;
        <ec2, En.parle, El1.ecoute, El2.ecoute>;
        <ec1, En.parle, El0.ecoute>;
        <ec1, En.parle, El1.ecoute>;
        <ec1, En.parle, El2.ecoute>;
        <ec0, En.parle>;            // cela s'appelle radoter
        <p2, En.polycop, El0.ecrit, El1.ecrit>;
        <p2, En.polycop, El0.ecrit, El2.ecrit>;
        <p2, En.polycop, El1.ecrit, El2.ecrit>;
        <p1, En.polycop, El0.ecrit>;
        <p1, En.polycop, El1.ecrit>;
        <p1, En.polycop, El2.ecrit>;
        <p0, En.polycop>;           // et cela gaspiller du papier
edon
```

3.7.2 Objectifs

La modélisation des priorités peut s'avérer difficile, mais surtout l'énumération des synchronisations est fastidieuse. Le concept de diffusion AltaRica peut-être vu comme une *macro* remplaçant cette énumération par une déclaration d'événements *optionnels* dans la synchronisation.

3.7.3 Un exemple

Le modèle suivant utilisant la diffusion répond également à la question 3.9.

```
node Eleve
  state present, crayon : bool;
  event in_out, ecoute, ecrit;
  trans true |- in_out -> present := ~present, crayon := true;
        true |- in_out -> present := ~present, crayon := false;
        present |- ecoute -> ;
        present & crayon |- ecrit -> ;
edon

node Enseignant
  state present : bool;
  event in_out, parle, polycop;
  trans true |- in_out -> present := ~present;
        present |- parle, polycop -> ;
edon

node CoursDifficile
  sub   En : Enseignant;
        El0, El1, El2 : Eleve;
  sync  <En.parle, El0.ecoute?, El1.ecoute?, El2.ecoute?>;
        <En.polycop, El0.ecrit?, El1.ecrit?, El2.ecrit?> <= 2;
edon
```

3.7.4 Idée intuitive de la sémantique

Les événements suffixés d'un `?` sont optionnels. Plus précisément, ils sont obligatoires si ils sont possibles. Par contre s'ils sont impossibles, ils n'empêchent pas les autres événements du vecteur de synchronisation d'avoir lieu.

L'indication `<= 2` dans le second vecteur indiquent qu'au maximum 2 événements optionnels peuvent avoir lieu. Cela permet la modélisation d'une gestion de ressources.

Si les trois élèves sont présents avec chacun un crayon, l'enseignant va devoir choisir les deux élèves qui auront un polycopie. Ce choix sera non-déterministe, et cela explique pourquoi les priorités entre les événements constituent un ordre partiel et non un ordre total.

3.8 Visibilité et portée des *objets* en AltaRica

3.8.1 Visibilité par défaut

	Propriétaire	Frères	Père	(Grand-Père)*
state	Garde Affectation Assertion			
flow	Garde Assertion		Garde Assertion	
event	Transition Synchronisation		Synchronisation	

3.8.2 Quelques problèmes

Pour pouvoir contrôler un sous-nœud en fonction de son état, il est nécessaire de *dupliquer* l'état dans une variable de flux. De même, si un flux d'un composant est un simple *observateur* que l'on souhaite voir au plus haut niveau hiérarchique, il est nécessaire de le *dupliquer* dans tous les nœuds intermédiaires. Par exemple :

```
node   Interrupteur
state  ouvert : bool;
flow   f1, f2 : [0,1];
       f_ouvert : bool;        // pour pouvoir controler
event  push;
trans  true |- push -> ouvert := ~ouvert;
assert ~ouvert => (f1 = f2);
       ouvert = f_ouvert;
edon

node Lampe
flow allume : bool;
edon

node Circuit
sub    I : Interrupteur;
       L : Lampe;
flow   allume : bool;         // pour propager l'observation
assert I.f_ouvert => (I.f2=0); // sens du montage
       allume = L.allume;
edon

node VisibiliteDefautStateFlow
```

```
sub    C : Circuit;
flow   couleur : {rouge, vert};
assert C.allume = (couleur=rouge);
edon
```

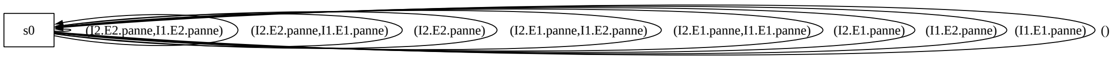
Dans la plus part des systèmes, du logiciel contrôle des composants pouvant tomber en panne. Les défaillances sont souvent des événements indépendants, et synchroniser deux pannes indépendantes a peu de sens. L'écriture en AltaRica semble donc être :

```
node   Equipement
event  panne;
trans  true |- panne -> ;
edon

node Inter
sub  E1, E2 : Equipement;
edon

node VisibilitePbEvent
sub    I1, I2 : Inter;
edon
```

En fait, la sémantique du nœud `VisibilitePbEvent` est la suivante :



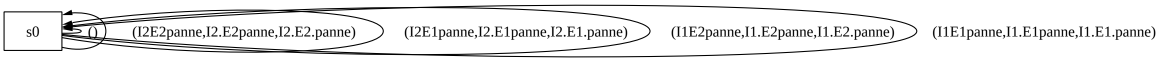
En AltaRica, afin d'éviter ce problème dès qu'il y a plus de deux niveaux hiérarchique, il est nécessaire de *renommer* artificiellement par synchronisation les pannes dans tous les nœuds intermédiaires. Il faut donc écrire :

```
node   Equipement
event  panne;
trans  true |- panne -> ;
edon

node Inter
sub    E1, E2 : Equipement;
event  E1panne, E2panne;
trans  true |- E1panne, E2panne -> ;
sync   <E1panne, E1.panne>;
       <E2panne, E2.panne>;
edon

node VisibiliteDefautEvent
sub    I1, I2 : Inter;
event  I1E1panne, I1E2panne, I2E1panne, I2E2panne;
trans  true |- I1E1panne, I1E2panne, I2E1panne, I2E2panne -> ;
sync   <I1E1panne, I1.E1panne>;
       <I1E2panne, I1.E2panne>;
       <I2E1panne, I2.E1panne>;
       <I2E2panne, I2.E2panne>;
edon
```

afin d'avoir la sémantique voulue, à savoir :



3.8.3 Modification des visibilités par attributs

Afin de faciliter l'écriture des modèles AltaRica précédents, des attributs permettant de changer la visibilité et la portée par défaut sont autorisés. En **gras**, la sémantique par défaut d'attribut.

	Attribut	Propriétaire	Frères	Père	(Grand-Père)*
state	**private**	Garde Affectation Assertion			
	parent	Garde Affectation Assertion		Garde Assertion	
	public	Garde Affectation Assertion		Garde Assertion	Garde Assertion
flow	private	Garde Assertion			
	parent	Garde Assertion		Garde Assertion	
	public	Garde Assertion		Garde Assertion	Garde Assertion
event	**parent**	Transition Synchronisation		Synchronisation	
	public	Transition Synchronisation		Synchronisation	Synchronisation *si non déjà synchronisé*

Avec cette extension du langage, les solutions aux problèmes précédents deviennent pour les variables :

```
node   Interrupteur
state  ouvert : bool : parent;       // pour pouvoir controler
flow   f1, f2 : [0,1];
event  push;
trans  true |- push -> ouvert := ~ouvert;
assert ~ouvert => (f1 = f2);
edon

node Lampe
flow allume : bool : public;         // pour propager l'observation
edon

node Circuit
sub    I : Interrupteur;
       L : Lampe;
assert I.ouvert => (I.f2=0);         // sens du montage
edon

node VisibiliteStateFlow
sub    C : Circuit;
flow   couleur : {rouge, vert};
assert C.L.allume = (couleur=rouge);
edon
```

et pour les événements :

```
node   Equipement
event  panne : public;        // pour propager l'événement
trans  true |- panne -> ;
edon

node Inter
sub    E1, E2 : Equipement;
edon

node VisibiliteEvent
sub    I1, I2 : Inter;
edon
```

3.9 Le modèle de calcul AltaRica

En réalité, le modèle de calcul AltaRica est plus complexe que celui présenté en début de chapître.

Definition 3.2 *Un automate à contraintes* AltaRica *est un sextuplet :*
- *un ensemble fini d'automates à contraintes* AltaRica $\vec{AC}$
- *un ensemble fini de variables de d'états $\vec{s}$,*
- *un ensemble fini de variables de flux $\vec{f}$,*
- *un ensemble fini d'événements E,*
- *un ensemble fini de transitions :*

$$G(\vec{s}, \vec{f}, \vec{AC}.f) \quad \xrightarrow{e} \quad \vec{s} := \sigma(\vec{s}, \vec{f}, \vec{AC}.f)$$

  $G(\vec{s}, \vec{f}, \vec{AC}.f)$ est un(e) garde (c-à-d une expression booléenne) et $e \in E$.
- *une assertion $A(\vec{s}, \vec{f}, \vec{AC}.f)$*
- *un ordre partiel $\prec_E$ pour des priorités.*
- *un ensemble fini de vecteur de synchronisation muni d'un ordre partiel. Un vecteur de synchronisation est un élement de $(E \cup \epsilon) \times \prod(AC.E \cup \epsilon)$.*

Tous les ensembles finis peuvent être vides. Les feuilles ont notamment un ensemble vide d'automates à contraintes.

La sémantique d'un automate peut-être obtenue par une mise à plat qui produit un automate à contraintes AltaRica dont l'ensemble $\vec{AC}$ est **vide**.

Chapitre 4

Quelques modèles de conception AltaRica

4.1 Le

Chapitre 5

La vérification de modèles AltaRica

5.1 La vérification de modèles

Un vérificateur de modèle est un outil qui prend en entrée : d'une part un modèle ; d'autre part une formule logique, pour fournir comme résultat : soit `VRAI`, soit `FAUX` avec généralement un contre-exemple.

Il existe deux vérificateurs de modèles AltaRica :

`acheck` **(qui fait partie des `altatools`)** est basé sur une représentation explicite par un graphe de l'ensemble des comportements possibles. Les propriétés sont exprimées à partir de propriétés élémentaires (expression booléenne et étiquettes de transition), d'opérateurs ensemblistes, d'opérateurs pour passer des formules d'états aux formules de transitions et réciproquement, d'opérateurs de points fixes sans imbrication, et des opérateurs spécifiques `loop` et `trace`.

`MecV` est basé sur une représentation implicite par BDD (Binary Decision Diagram) de l'ensemble des comportements possibles. Les propriétés sont exprimées par des relations constructibles à partir de relations élémentaires (expression booléenne et étiquettes de transition), d'opérateurs ensemblistes, d'opérateurs de points fixes **avec imbrications**.

Nous utiliserons cette année `acheck`.

5.2 Présentation de `acheck`

Distribution et commentaires sur le manuel de `acheck`.

Ce manuel est disponible à l'adresse : (`http ://altarica.labri.fr/Tools/Acheck/acheck.ps`).

Deuxième partie

Le langage B