Conceptions formelles à l'aide de vérificateurs de modèles

A. Griffault

FAC 2007, Toulouse. 

15 mars 2007

Plan

Plan

Les concepts généraux

Objectif et moyen

- Pouvoir raisonner sur les logiciels et les systèmes afin de connaître leurs comportements et les contrôler.
- Les systèmes sont des objets mathématiques.

Processus

- Obtenir un modèle formel du logiciel ou du système.
- L'analyser par une technique formelle.
- Transposer les résultats des modèles aux systèmes réels.

Les problèmes

- Le modèle est-il fidèle ? *validation*.
- Peux t-on tout vérifier ? *décidabilité*.
- La transposition est-elle toujours possible ? *abstraction*

Les différentes techniques formelles

Les démonstrateurs de théorèmes

- Ils fabriquent des preuves **justes**.
- Un programme est vu comme une fonction de calcul, (terminaison, résultat calculé est bien celui attendu).

Les différentes techniques formelles

L'interprétation abstraite

- 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.
- Implication dans un seul sens.

Les différentes techniques formelles

Les langages dédiés

- Un langage avec des primitives dédiées par type d'application.
- Un compilateur vers un langage généraliste (C ; ADA).

Les différentes techniques formelles

Les vérificateurs de modèles

- Décider si un modèle d'un système satisfait ou non des propriétés logiques comportementales.
- Besoin de "valider" le modèle.
- Problème de "l'explosion combinatoire".

Les différentes techniques formelles

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.
- Utilise souvent un démonstrateur de théorèmes.

Les différentes techniques formelles

Les générateurs de tests

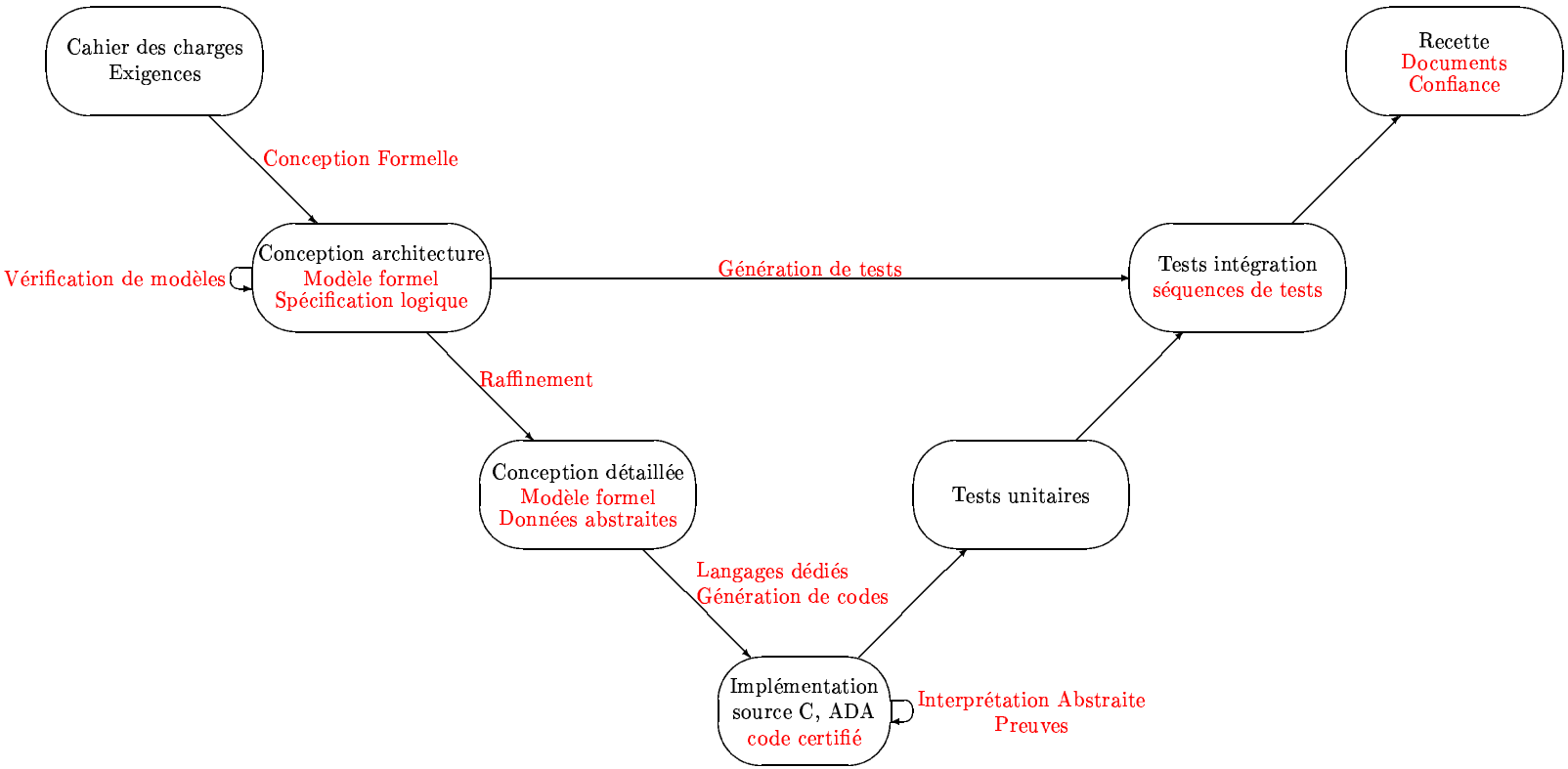
- Générer des séquences de tests à partir d'un modèle de la spécifications.
- Tests relatifs à des objectifs de tests spécifiques.
- Permet de "valider" le système réel sans le voir.

Les différentes techniques formelles

La simulation stochastique

- Les processus stochastiques ajoutent dans la description des systèmes des grandeurs probabilistes.
- Calculer des probabilités relatives à des propriétés.
- Domaine privilégiée de la sûreté de fonctionnement.

Cycle de vie et techniques formelles



Plan

La vérification formelle de modèles

Le principe

- Le système est représenté par un modèle M,
- Les exigences sont décrites par une propriété ϕ,
- Un vérificateur de modèles prend en entrée d'une part M, d'autre part ϕ et répond **automatiquement** à la question $M \models \phi$? De plus, il peut fournir une explication lorsque la réponse est négative.

$\rightarrow$ nombreux outils académiques disponibles.

La vérification formelle de modèles

Les limites théoriques

Les couples (classe de modèles, logique) pour lesquels le problème est décidable.

- Systèmes finis : toutes les logiques comportementales sont décidables.
- Systèmes infinis : presque rien n'est décidable $\rightarrow$ études de systèmes infinis particuliers (RdP, avec compteurs, FIFO, temporisés) pour lesquels des fragments de logiques sont décidables.

$\rightarrow$ nombreux travaux de recherches théoriques.

La vérification formelle de modèles

Les limites pratiques

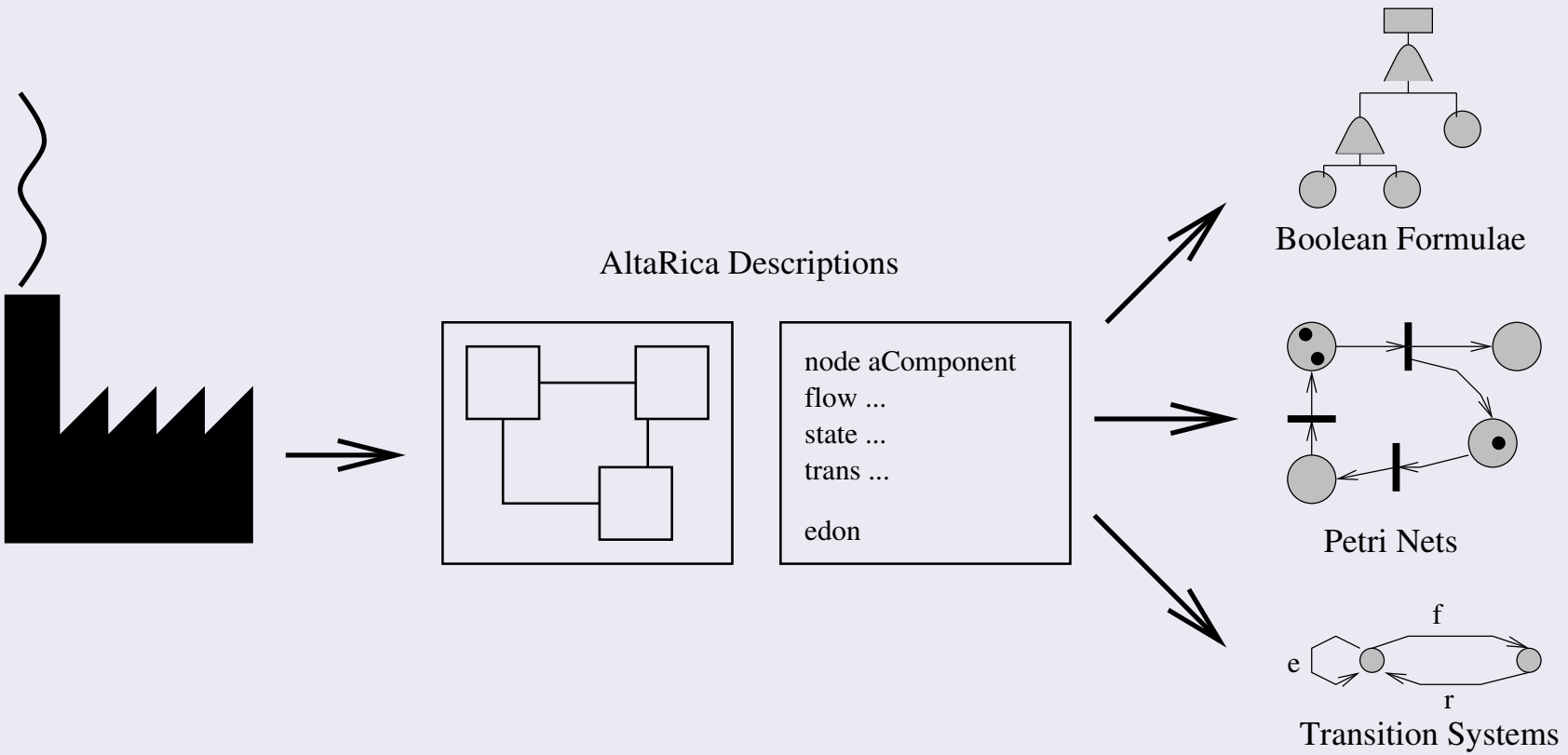
- Les machines et par les techniques utilisées pour coder le graphe en mémoire (explicite, BDD, ordre partiel, DBM ...). Actuellement de l'ordre de 100 variables booléennes.
- La validité du modèle M.
- La difficulté de l'écriture de ϕ.

$\rightarrow$ nombreux travaux de recherches algorithmiques et méthodologiques.

Plan

Le projet AltaRica

Les objectifs



Le projet AltaRica

Les étapes

- 97/99 : définition, consolidation du langage et outillage.
- Aujourd'hui, quatre lignes d'outils et des collaborations :

  OCAS AltaRica Dassault (simulation, arbre de défaillance)
  SIMFIA V2 Airbus (simulation, arbre de défaillance)
  AltaRica DF A. Rauzy (stochastique, arbre de défaillance)
  arc, mec LaBRI (sémantique et vérification).
  T_AltaRica (IRCCyN).
  AltaRica↔Lustre (CERT ONERA).
  AltaRica + Types Abstraits (ACI Projet Persée).

Le modèle de calcul AltaRica

les automates à contrainte

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

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

  $G(\vec{s})$ 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 concepts fondamentaux du langage AltaRica

Modularité et compositionnabilité

- La hiérarchie.
- Les interfaces (visibilité) des composants.
- Description d'un noeud :
  - La distinction variable d'état/variable de flux.
  - Les transitions.
  - Les assertions : flux et états.
- Description des interactions entre composants :
  - Les assertions : flux, états et interfaces visibles.
  - Les vecteurs de synchronisation généralisés.
  - Les priorités entre événements.
- Bisimulation.

Le contrôle du niveau d'une cuve

Le cahier des charges

Le système que l'on souhaite concevoir est composé :

- d'un réservoir contenant **toujours** suffisamment d'eau pour alimenter l'exploitation,
- d'une cuve,
- de deux canalisations amont reliant le réservoir à la cuve, et permettant d'amener l'eau à la cuve,
- d'une canalisation aval permettant de vider l'eau de la cuve,
- chaque canalisation est équipée d'une vanne commandable, afin de réguler l'alimentation et la vidange de la cuve,
- d'un contrôleur.

Le contrôle du niveau d'une cuve

Les spécifications

1. Le niveau de la cuve doit rester dans les zones intermédiaires.
2. En fonction du nombre de vannes en panne, et après une éventuelle période initiale la plus courte possible, le débit de la vanne aval doit être le plus grand possible.

Une vanne

La description technique

- Trois niveaux de débits : 0, 1 et 2.
- Deux signaux de commande : `inc` et `dec`
- La rouille avec le temps peux bloquer la vanne. Le débit n'est plus modifiable

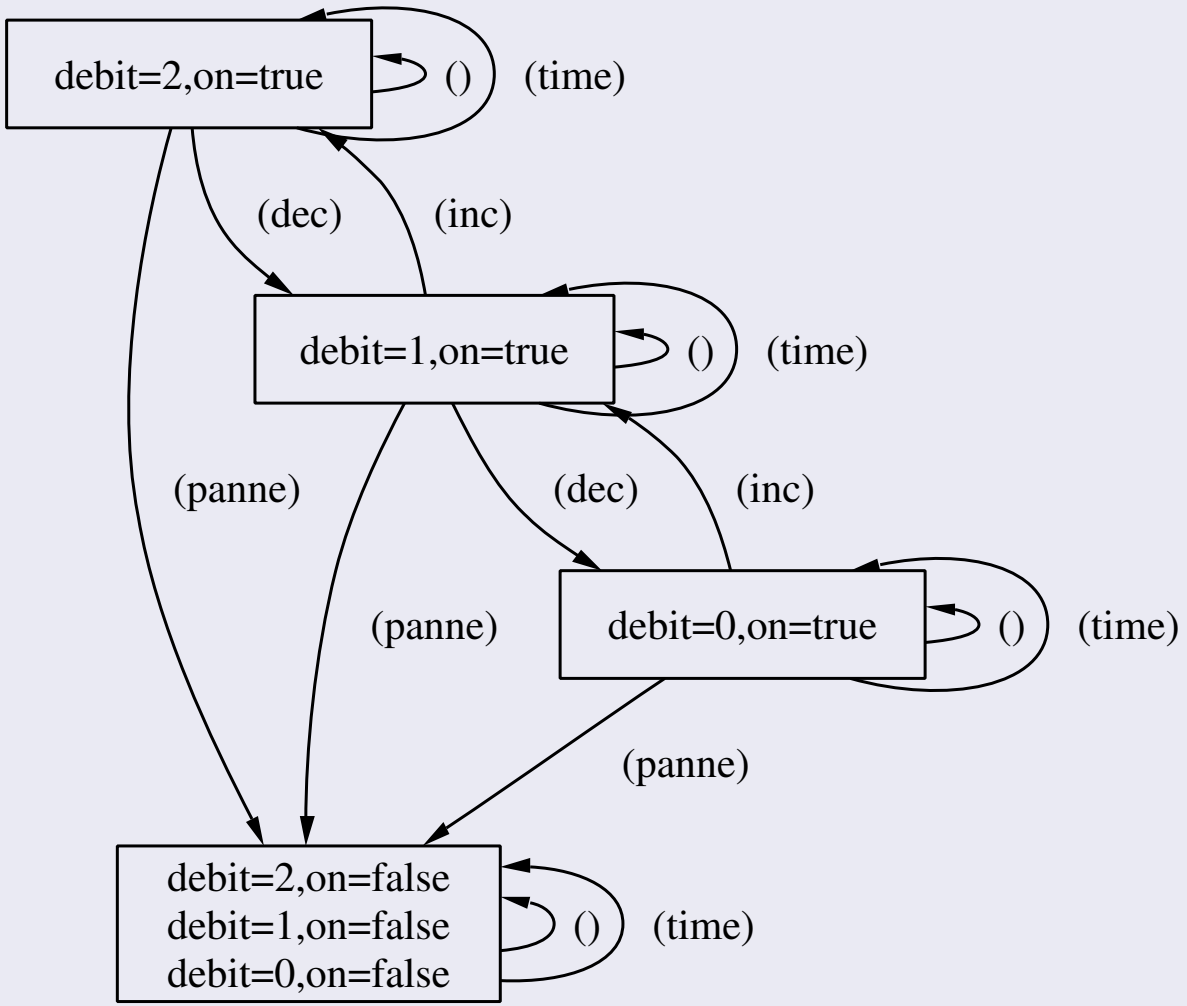
Une vanne

Le code AltaRica

```
node Vanne
  state
    debit : [0,2] : public;
    on : bool : public;
  event
    inc, dec, panne, time;
  trans
    on   |- inc -> debit := debit+1;
    on   |- dec -> debit := debit-1;
    on   |- panne -> on := false;
    true |- time ->;
  init
    on := true, debit := 0;
edon
```

Une vanne

La sémantique AltaRica



La cuve

La description technique

- Quatre capteurs qui délimitent 5 zones (de 0 à 4).
- Un débit maximal amont de 4.
- Un débit maximal aval de 2.
- Des régles d'évolution du niveau.
  - si $amont > aval$ alors dans le futur, le niveau augmentera de 1.
  - si $amont < aval$ alors dans le futur, le niveau diminuera de 1.
  - si $amont = aval = 0$ alors le niveau ne changera pas.
  - si $amont = aval > 0$ alors le futur est imprévisible. Le niveau pourra augmenter de 1, diminuer de 1 ou bien rester inchangé.

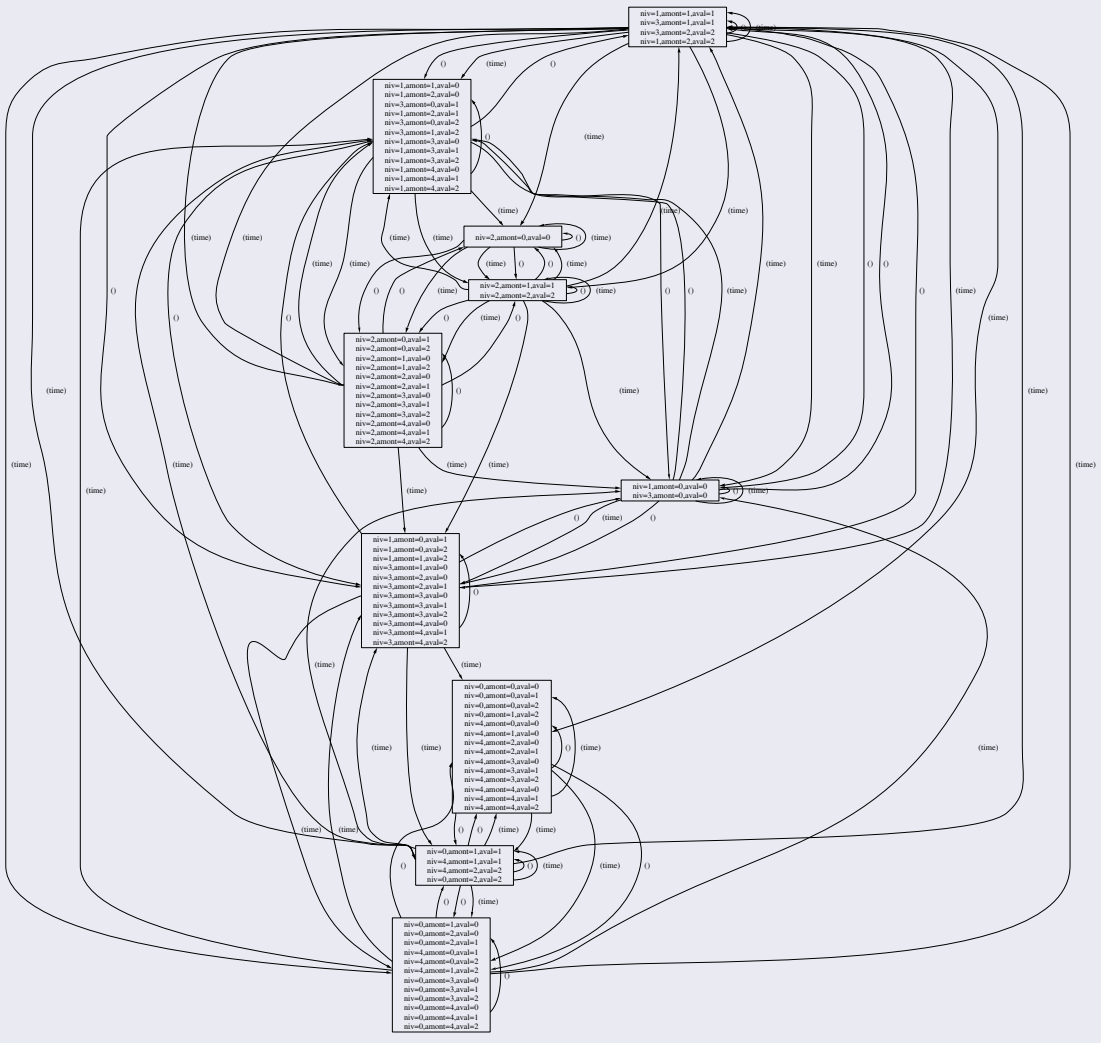
La cuve

Le code AltaRica

```
node Cuve
  flow    amont : [0,4];
          aval  : [0,2];
  state   niv : [0,4] : public;
  event   time;
  trans
    (amont=aval)                |- time -> ;
    (amont=aval) & (aval>0)     |- time -> niv := niv+1;
    (amont=aval) & (aval>0)     |- time -> niv := niv-1;
    (amont>aval)                |- time -> niv := niv+1;
    (amont>aval) & (niv=4)      |- time -> ;
    (amont<aval)                |- time -> niv := niv-1;
    (amont<aval) & (niv=0)      |- time -> ;
  init    niv := 2;
edon
```

La cuve

La sémantique AltaRica



Le contrôleur

La description technique

Il commande les vannes avec les objectifs de la spécification.

1. Le niveau de la cuve ne doit jamais atteindre les zones 0 ou 4.
2. Le débit de la vanne aval doit être le plus important possible.

Un contrôleur très permissif

- Il choisit les commandes aléatoirement.

Le contrôleur

Le code AltaRica

```
node ControleurPermissif
  event nop, inc1, inc2, inc3, dec1, dec2, dec3,
    inc1inc2,inc1inc3,inc1dec2,inc1dec3,inc2inc3,inc2dec1,
    inc2dec3,inc3dec1,inc3dec2,dec1dec2,dec1dec3,dec2dec3,
    inc1inc2inc3, inc1inc2dec3, inc1dec2inc3, inc1dec2dec3,
    dec1inc2inc3, dec1inc2dec3, dec1dec2inc3, dec1dec2dec3;
  flow    d1, d2, d3 : [0,2]; // les debits des vannes
          n : [0,4];          // le niveau de la cuve
          v1, v2, v3 : bool;  // les pannes des vannes
  trans
  true |- nop, inc1, inc2, inc3, dec1, dec2, dec3,
    inc1inc2,inc1inc3,inc1dec2,inc1dec3,inc2inc3,inc2dec1,
    inc2dec3,inc3dec1,inc3dec2,dec1dec2,dec1dec3,dec2dec3,
    inc1inc2inc3, inc1inc2dec3, inc1dec2inc3, inc1dec2dec3,
    dec1inc2inc3, dec1inc2dec3, dec1dec2inc3, dec1dec2dec3
    -> ;
edon
```

Le système

La description technique

Les hypothèses (assez classiques) :

- Les commandes ne prennent pas de temps.
- Entre deux pannes et/ou cycle *temporel*, le contrôleur à toujours le temps de donner au moins un ordre.
- le système à toujours le temps de réagir entre deux commandes.

Le système

La hierarchie et le contrôle

```
node System
  sub    V1, V2, V3 : Vanne; Cu : Cuve;
         Co : ControleurPermissif;
  state  controle : bool;    init controle := true;
  flow nbPannes : [0,3];
  event  commande, time;
  trans  controle  |- commande -> controle := false;
         ~controle |- time     -> controle := true;
  assert (nbPannes = 0) = (V1.on & V2.on & V3.on);
         (nbPannes = 1) = ((V1.on & V2.on & ~V3.on)|
                           (V1.on & ~V2.on & V3.on)|
                           (~V1.on & V2.on & V3.on));
         (nbPannes = 2) = ((V1.on & ~V2.on & ~V3.on)|
                           (~V1.on & V2.on & ~V3.on)|
                           (~V1.on & ~V2.on & V3.on));
         (nbPannes = 3) = (~V1.on & ~V2.on & ~V3.on);
```

Le système

Les variables partagées

```
assert
  // Les canalisations
  Cu.amont = V1.debit+V2.debit;
  Cu.aval  = V3.debit;
  // Pour observer les debits des vannes
  Co.d1 = V1.debit;
  Co.d2 = V2.debit;
  Co.d3 = V3.debit;
  // Pour observer le niveau de la cuve
  Co.n  = Cu.niv;
  // Pour observer les pannes des vannes
  Co.v1 = V1.on;
  Co.v2 = V2.on;
  Co.v3 = V3.on;
```

Le système

Les communications

```
  sync
    // les actions du controleur
    <commande, Co.inc1, V1.inc>;
...
    <commande, Co.inc1dec2, V1.inc, V2.dec>;
...
    <commande, Co.inc1inc2dec3, V1.inc, V2.inc, V3.dec>;
...
    <commande, Co.nop>;
    // les actions du systeme
    // les pannes ont lieu pendant un cycle
    <time, Cu.time, V1.panne, V2.time, V3.time>;
    <time, Cu.time, V1.time, V2.panne, V3.time>;
    <time, Cu.time, V1.time, V2.time, V3.panne>;
    // Le temps s'ecoule normalement
    <time, Cu.time, V1.time, V2.time, V3.time>;
edon
```

Le système

La sémantique

```
/*
 * # state properties : 2
 *
 * initial = 1
 * any_s = 1861
 *
 * # transition properties : 5
 *
 * epsilon = 1861
 * any_t = 9334
 * self_epsilon = 1861
 * self = 1861
 * not_deterministic = 1300
 */
```

Plan

Validation et Vérification

La validation

Il faut s'assurer que le modèle est "réaliste".

- États puits ?
- Forte connexité du graphe ?
- Comportements instables ?
- ...
- Simulation.

La vérification

- Propriétés de sûreté : faciles sur un graphe fini.
- Propriétés de vivacité, d'équité : liées aux propriétés de fortes connexité dans un graphe fini.

Validation et Vérification

Le code AltaRica

```
with System
do
 show(all)                    >  '$NODENAME.res';
 dead          := any_s - src(any_t - self_epsilon);
 notCFC        := any_t - loop(any_t,any_t);
 niveau0       := [Cu.niv = 0];
 niveau4       := [Cu.niv = 4];
 nonControle   := label time;
 controle      := label commande;
 ER            := niveau0 | niveau4 | dead;
 out2          := [V3.debit=2];
 controleUp    := controle & label V3.inc;
 controleDown  := controle & label V3.dec;
 cfcInstable   := loop(any_t,nonControle | (epsilon-self_epsilon));
 // sortie des resultats
 show(all)                    >  '$NODENAME.resVV';
 test(dead,0)                 >  '$NODENAME.propVV';
 test(notCFC,0)               >>  '$NODENAME.propVV';
 test(ER,0)                   >>  '$NODENAME.propVV';
 test(cfcInstable,0)          >>  '$NODENAME.propVV';
done
```

Validation et Vérification

Les résultats : propriétés d'états

```
/*
 * # state properties : 7
 *
 * out2 = 650
 * initial = 1
 * dead = 0
 * any_s = 1861
 * niveau0 = 224
 * niveau4 = 393
 * ER = 617
 */
```

Validation et Vérification

Les résultats : propriétés de transitions

```
/*
 * # transition properties : 11
 *
 * controleUp = 821
 * controleDown = 856
 * cfcInstable = 0
 * epsilon = 1861
 * any_t = 9334
 * self_epsilon = 1861
 * notCFC = 2661
 * nonControle = 3313
 * controle = 4160
 * self = 1861
 * not_deterministic = 1300
 */
```

Synthèse de contrôleurs

Le problème

- Est-il possible de contrôler ?
- Si oui, comment ?

Avec un vérificateur

- Mettre un contrôleur “permissif” dans le modèle.
- Calculer le plus grand contrôleur sûr. (Ramadge & Wonham).
- “Projeter” le contrôleur sur le contrôleur permissif.

Synthèse de contrôleurs

L'idée de la spécification

```
/* Les equations classiques
 *  - de stratégies gagnantes
 *  - et de synthèse de contrôleurs
 *  coupA représente les actions du joueur A ou du controleur
 *  coupB représente les actions du joueur B ou du système
 *  Gagne représente des configurations de A
 *    - les configurations à atteindre lors d'un pppf
 *    - le complémentaire des configurations à éviter lors d'un pgpf
 *  Perdu représente des configurations de B
 *    - les configurations à atteindre lors d'un pppf (pour que A soit gagnant)
 *    - les configurations à éviter lors d'un pgpf (pour que A soit gagnant)
 Gagnant                = Gagne |& \\ | pour pppf et & pour pgpf
                          src(CoupGagnant);
 CoupGagnant            = coupA & rtgt(Perdant);
 Perdant                = Perdu |& \\ | pour pppf et & pour pgpf
                          (src(CoupPerdant) - src(CoupNonPerdant));
 CoupPerdant            = coupB & rtgt(Gagnant);
 CoupNonPerdant         = coupB - rtgt(Gagnant);
*/
```

Synthèse de contrôleurs

La spécification des contrôleurs

```
with System
do
 // PREMIER CONTROLEUR (le moins optimal)
 // les actions de controle pour eviter ER : pgpf
 GagneER := src(controle) - ER;
 PerduER := src(nonControle) - ER;
 CtrlER  -= controle &
            rtgt(PerduER &
                 (src(nonControle & rtgt(GagneER & src(CtrlER)))-
                  src(nonControle - rtgt(GagneER & src(CtrlER)))));

 // AUTRE CONTROLEUR (le meilleur possible)
 // les actions de controle pour rester dans out2 et pour eviter ER : pgpf
 GagneOut2ER := (src(controle) & out2) - ER;
 PerduOut2ER := (src(nonControle) & out2) - ER;
 CtrlOut2ER  -= controle &
                rtgt(PerduOut2ER &
                     (src(nonControle & rtgt(GagneOut2ER & src(CtrlOut2ER)))-
                      src(nonControle - rtgt(GagneOut2ER & src(CtrlOut2ER)))));
 // les actions de controle pour atteindre CtrlOut2ER : pppf
 GagnePreOut2ER := src(CtrlOut2ER) - ER;
 PerduPreOut2ER := tgt(nonControle) - ER;
 // calcul d'un preambule le plus optimal possible
 CtrlPreOut2ER += controleUp &
       rtgt(PerduPreOut2ER |
            (src(nonControle & rtgt(GagnePreOut2ER | src(CtrlPreOut2ER)))-
             src(nonControle - rtgt(GagnePreOut2ER | src(CtrlPreOut2ER)))));
done
```

Synthèse de contrôleurs

Le calcul des contrôleurs

```
with System
do
 // Le système est-il controlable
 ControlableER          := initial & src(CtrlER);
 ControlablePreOut2ER   := initial & src(CtrlPreOut2ER);
 // les erreurs de commande possibles
 ErreurControleER          := controle - CtrlER;
 ErreurControleOut2ER   := controle - (CtrlOut2ER | CtrlPreOut2ER);
 // Génération des trois controleurs
 project(any_s, CtrlER, 'ControleurOut012_$NODENAME', true, Co)
> 'Alt/ControleurOut012_$NODENAME.alt';
 project(any_s, CtrlOut2ER|CtrlPreOut2ER, 'ControleurOut2_$NODENAME', true, Co)
> 'Alt/ControleurOut2_$NODENAME.alt';
 // sortie des resultats
 show(all)                          >  '$NODENAME.resSC';
 test(ControlableER,1)              > '$NODENAME.propSC';
 test(ErreurControleER,0)           >> '$NODENAME.propSC';
 test(ControlablePreOut2ER,1)       >> '$NODENAME.propSC';
 test(ErreurControleOut2ER,0)       >> '$NODENAME.propSC';
done
```

Synthèse de contrôleurs

Les résultats : propriétés d'états

```
/*
 * # state properties : 15
 *
 * out2 = 650
 * GagneER = 617
 * PerduER = 627
 * GagneOut2ER = 213
 * PerduOut2ER = 213
 * GagnePreOut2ER = 0
 * PerduPreOut2ER = 617
 * initial = 1
 * dead = 0
 * any_s = 1861
 * niveau0 = 224
 * niveau4 = 393
 * ER = 617
 * ControlableER = 1
 * ControlablePreOut2ER = 0
 */
```

Synthèse de contrôleurs

Les résultats : propriétés de transitions

```
/*
 * # transition properties : 16
 *
 * controleUp = 821
 * controleDown = 856
 * cfcInstable = 0
 * epsilon = 1861
 * CtrlER = 81
 * CtrlOut2ER = 0
 * CtrlPreOut2ER = 0
 * any_t = 9334
 * self_epsilon = 1861
 * notCFC = 2661
 * nonControle = 3313
 * controle = 4160
 * ErreurControleER = 4079
 * ErreurControleOut2ER = 4160
 * self = 1861
 * not_deterministic = 1300
 */
```

Synthèse de contrôleurs

Les contrôleurs générés : extrait de l'optimal pour 0 panne

```
/*
 * This node is the result of the projection of the node 'SystemP0'
 * on its subnode 'Co'.
 */
...
   false |- inc2dec3 -> ;
   n = 1 and (d3 = 0 and (d2 = 0 and d1 = 2 or d2 = 1 and (d1 = 1 or d1 = 2)
   or d2 = 2 and d1 = 1) or d3 = 1 and d2 = 2 and d1 = 2) or n = 2 and
   (d3 = 0 or d3 = 1) and (d2 = 0 and d1 = 2 or (d2 = 1 or d2 = 2) and
   (d1 = 1 or d1 = 2)) or (n = 3 or n = 4) and (d3 = 0 and (d1 = 1 or d1 = 2)
   or d3 = 1 and (d2 = 0 and (d1 = 1 or d1 = 2) or d2 = 1 and d1 = 1))
   |- inc3dec1 -> ;
   n = 1 and (d3 = 0 and (d2 = 1 and (d1 = 1 or d1 = 2) or d2 = 2 and
   (d1 = 0 or d1 = 1)) or d3 = 1 and d2 = 2 and d1 = 2) or n = 2 and
   (d3 = 0 or d3 = 1) and (d2 = 1 and (d1 = 1 or d1 = 2) or d2 = 2) or
   (n = 3 or n = 4) and (d3 = 0 and (d2 = 1 or d2 = 2) or d3 = 1 and
   (d2 = 1 and (d1 = 0 or d1 = 1) or d2 = 2 and d1 = 0))
   |- inc3dec2 -> ;
...
```

Bisimulation et simulations

Définition et propriété

- (q, q') sont bisimilaires si quelque soit le choix fait par l'un, l'autre peut faire un choix qui le conduit dans un état équivalent par bisimulation.
- deux états bisimilaires sont indiscernables par une propriété du μ-calcul.
- deux systèmes sont bisimilaires ssi leurs états initiaux sont bisimilaires.

Bisimulation et simulations

L'expression logique de la bisimulation

```
bisim(s,s') -= eqC(s,s') &
  ([e][t](transA(s,e,t) =>
    <e'><t'>(transA'(s',e',t') & bisim(t,t')))) &
  ([e'][t'](transA'(s',e',t') =>
    <e><t>(transA(s,e,t) & bisim(t,t'))));

:rel-card bisim >> bisim.res

isBisim(x) :=
  x = (([a] (initA(a) =>
        <a'> (initA'(a') & bisim(a,a')))) &
       ([a'] (initA'(a') =>
        <a> (initA(a) & bisim(a,a'))))));

:display isBisim >> bisim.res
```

Bisimulation et simulations

Simulation, Implémentation et Raffinement

```
simAA'(s,s') -= eqC(s,s') &
  ([e][t](transA(s,e,t) =>
    <e'><t'>(transA'(s',e',t') & simAA'(t,t'))));
:rel-card simAA' >> bisim.res
isSimAA'(x) :=
  x = ([a] (initA(a) => <a'> (initA'(a') & simAA'(a,a'))));
:display isSimAA' >> bisim.res

simA'A(s,s') -= eqC(s,s') &
  ([e'][t'](transA'(s',e',t') =>
    <e><t>(transA(s,e,t) & simA'A(t,t'))));
:rel-card simA'A >> bisim.res
isSimA'A(x) :=
  x = ([a'] (initA'(a') => <a> (initA(a) & simA'A(a,a'))));
:display isSimA'A >> bisim.res
```

Bisimulation et simulations

Un exemple

```
node DeuxVannes
  sub     V1, V2 : Vanne
  flow    debit : [0,2] : public;
          on : bool : public;
  event   inc, dec, panne, time;
  trans   true |- inc, dec, panne, time -> ;
  sync    <time, V1.time, V2.time>;
          <panne, V1.panne>;
          <panne, V2.panne>;
          <inc, V1.inc?, V2.inc?> >=1;
          <dec, V1.dec?, V2.dec?> >=1;
  assert (V1.debit < V2.debit) => (debit = V1.debit) ;
          (V1.debit >= V2.debit) => (debit = V2.debit) ;
          on = (V1.on | V2.on);
edon
```

Bisimulation et simulations

Résultats

```
cardinal of Vanne!t: 19
cardinal of DeuxVannes!t: 148
cardinal of bisim: 15
(false)
cardinal of simAA': 15
(false)
cardinal of simA'A: 36
(true)
```

Plan

Conclusion

Bilan

- La vérification de modèle doit se limiter aux aspects contrôle.
- La “liste” des propriétés est peu longue. La difficulté d’écrire les propriétés en logiques comportementales est un faux problème.
- La véritable difficulté réside dans la tâche de modélisation. Le besoin de langages de description de systèmes est réel.
- La bisimulation est une propriété trop forte.
- Une simulation qui préserve une équivalence de trace est sans doute le raffinement utile en AltaRica.

Perspectives

- Un environnement AltaRica complet à terme.
- Plusieurs thèses en cours.