Vérification Formelle des modèles AltaRica
Formal Verification of AltaRica Models

Alain GRIFFAULT, Gérald POINT et Aymeric VINCENT
LaBRI - CNRS UMR 5800 - Université Bordeaux 1
351, cours de la Libération 33 405 Talence Cedex

Résumé

Le formalisme AltaRica a été créé au LaBRI en partenariat avec des industriels afin de pouvoir modéliser aisément des systèmes complexes. Doté d'une syntaxe concrète clairement définie dont la sémantique est parfaitement spécifiée, il permet à partir d'un même modèle d'utiliser plusieurs chaînes d'outils différentes afin d'en étudier différents aspects (arbres de défaillances, réseaux de Petri, ...). L'utilisation d'AltaRica au sein de projets d'envergure fait que de nombreux modèles existent et que des bibliothèques de composants AltaRica ont vu le jour.

Dans ce papier nous présentons l'utilisation du langage AltaRica dans le cadre de la vérification formelle. Après avoir rappeler les principaux concepts du langage ainsi que les principes de la vérification de modèles, nous présentons les résultats de l'étude des dysfonctionnements d'un cas industriel.

Summary

The AltaRica formalism has been designed at LaBRI within a partnership with french industrials. This language has been built in order to easy the modelling tasks of complex systems. Since its syntax and semantics are formally defined, the language is equipped with several chains of tools allowing the study of different aspects of systems using others formalisms (fault trees, Petri nets …). Today, there exist many AltaRica models or library of components.

The aim of this paper is to present the use of AltaRica in the model-checking framework. First we recall the main concepts of the AltaRica formalism and the principles of the model-checking, then we present some results about the dysfunctional analysis of an industrial case.

Introduction

Les méthodes d'analyse des systèmes critiques peuvent être classées en deux grandes familles suivant les aspects du système que l'on souhaite étudier. D'un côté les *méthodes formelles* qui s'attachent principalement aux aspects fonctionnels, tentent de mettrent en évidence les erreurs de conception d'un système considéré comme une fonction ou comme un programme. D'un autre côté, les méthodes issues de la *sûreté de fonctionnement* qui s'intéressent plutôt aux aspects dysfonctionnels, tentent d'évaluer de manière quantitative et qualitative les conséquences des pannes dans un système.

Face à la prolifération des modèles issus des différentes méthodes d'analyse utilisées, le LaBRI et un groupe d'industriels français ont développé le langage AltaRica[1][2]. L'objectif de ce langage est de fournir un cadre commun aux études tant fonctionnelles que dysfonctionnelles des partenaires du LaBRI. Récemment le langage a été doté d'outils de vérification de modèles, **Mec V**[6] et **acheck**, qui permettent d'étudier de manière exhaustive le graphe des comportements des systèmes modélisés en AltaRica. L'objectif de ce papier est de présenter l'utilisation des techniques de vérification de modèles sur le langage AltaRica, dans le cadre de l'étude des dysfonctionnements d'un système hydraulique.

La suite de cette présentation est organisée de la manière suivante. Tout d'abord, nous faisons un survol des méthodes formelles existantes. Nous rappelons ensuite les principales caractéristiques du langage AltaRica. Enfin nous terminons par les résultats d'une étude menée sur un système hydraulique de commande de vol d'un avion.

Les méthodes formelles

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.

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

- **La démonstration de théorèmes** utilise 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. 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.
- **La vérification de modèles** permet 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 possède de trop nombreuses données.
- **La génération de tests** utilise 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.

Le formalisme AltaRica

AltaRica est un langage de modélisation de haut niveau conçu pour faciliter la formalisation de systèmes complexes. Dans cette section nous présentons les principaux concepts de ce langage et l'intuition que l'on peut avoir de leur sémantique. Le lecteur intéressé par la définition formelle du langage pourra consulter les références sur le sujet [1] [2] [3].

Composants

Un système est décrit par un ensemble de composants appelés *nœuds*. Ils sont les éléments de base du modèle et peuvent représenter aussi bien un composant physique qu'une abstraction d'une partie du système étudié.
Un nœud AltaRica est une description syntaxique d'un *automate à contraintes* [4] qui comporte :

- Un ensemble de variables d'état : un état du composant est une affectation de ces variables.
- Un ensemble de variables de flux : chaque variable de flux représente une information émise et/ou reçue par le composant (p.ex. le débit d'un fluide, une tension, …). La valeur des variables de flux est contrainte par l'état du composant à l'aide d'assertions. Une affectation des variables de flux associée à un état du composant est appelé une *configuration*.
- Une assertion sur les variables d'état et de flux : il s'agit d'une contrainte logique sur les deux jeux de variables qui doit être vérifiée à tout instant par le composant. Cette contrainte permet d'exprimer d'une part la relation qui existe l'état du composant et ses flux et d'autre part de restreindre l'ensemble des configurations possibles du composant.
- Une ensemble d'événements : ils représentent soit des actions propres du composant (p.ex une panne) soit des actions de l'environnement sur le composant (p.ex. l'action d'un opérateur).
- Un ensemble de transitions : une transition représente un ensemble de changements d'état du composant. Elle est composée d'une condition, d'un événement et d'une affectation des variables d'état. La sémantique intuitive d'une transition est la suivante : lorsque l'événement a lieu, si la condition est vérifiée alors l'affectation des variables d'état peut être réalisée. Il est important de noter que seules les variables d'état sont modifiable par une transition ; les variables de flux sont mises à jour par l'assertion après la transition.

Ci-dessous est présenté un nœud AltaRica modélisant un interrupteur.

```
node Interrupt
  state open, stuck : bool ;
  flow f1, f2 : bool ;
  assert not open => (f1=f2);
  event open, close, failure ;
  trans
    not open and not stuck |- open -> open := true;
    open and not stuck |- close -> open := false;
    not stuck |- failure -> stuck := true;
  init open := true, stuck := false;
edon
```

Hiérarchie

Le langage AltaRica autorise des descriptions hiérarchiques des systèmes. Cette hiérarchie est introduite en déclarant dans un nœud englobant quels sont les sous-noeuds qui le composent. Un nœud hiérarchique (c.-à-d. qui contient des sous-nœuds) est plus qu'une simple boîte englobante car il est possible de lui associer des comportements comme pour un composant de base. De plus les comportements d'un nœud englobant peut contraindre ses sous-nœuds ; en particulier, son assertion est utilisée pour modéliser les éventuelles connexions de flux entre les sous-nœuds.
Ci-dessous est présenté un nœud hiérarchique AltaRica. Sur cet exemple, le nœud englobant contient deux interrupteurs *i1* et *i2*. L'assertion du nœud indique que les flux f1 de i1 et f2 de i2 sont égaux ; cette égalité étant vérifiée en tout état du noeud, on peut considérer que les flux i1.f1 et i2.f2 sont connectés.

```
node TwoInterrupts
  sub i1, i2 : Interrupt;
  assert i1.f2 = i2.f1;
edon
```

Synchronisations

Les assertions des noeuds hiérarchiques peuvent être considérées comme un mécanisme de synchronisation des variables flux. Le langage autorise un mécanisme semblable pour les événements. Ce mécanisme s'exprime en définissant une contrainte sur les occurrences des événements des sous-nœuds. Cette contrainte est un ensemble de vecteurs d'événements. Chaque vecteur indique que les événements qui le composent doivent être simultanés. Les seules combinaisons d'événements possibles dans un nœud hiérarchique sont les vecteurs décrits dans la contrainte de synchronisation.
L'exemple ci-dessous montre comment sont synchronisés les événements de deux interrupteurs afin que l'un se ferme lorsque l'autre s'ouvre :

```
node Main
  sub i1, i2 : Interrupt;
  sync <i1.open,i2.close>;
       <i1.close,i2.open>;
edon
```

Sur l'exemple précédent, l'événement *open* de *i1* doit nécessairement être synchrone avec l'événement *close* de *i2*. En conséquence, si l'un des deux interrupteurs est bloqué (à la suite de l'événement *failure*) alors la combinaison <i1.open, i2.close> ne pourra pas avoir lieu.
Le langage AltaRica autorise des synchronisations d'événements moins strictes qui ont été introduites pour modéliser les défauts de mode commun (voir [1] et [3] pour plus de détails).

Priorités

La notion de priorité exprime qu'un événement doit être traité de préférence avant un autre. Intuitivement, si à un instant donné deux événements sont possibles alors c'est le plus prioritaire qui a lieu. Par extension, dans un nœud hiérarchique, la définition de priorités sur les événements du nœud englobant induit une priorité sur ses vecteurs de synchronisation. Les priorités sont exprimées à l'aide d'une relation d'ordre partielle définie sur les événements d'un nœud ; il n'existe pas de priorité globale.

La vérification de modèles AltaRica

Généralement on regroupe sous le terme « vérification de modèles » (ou *model-checking*) l'ensemble des techniques qui reposent sur l'étude du graphe des comportements. Le système est modélisé dans un certain formalisme (automates à contraintes, automates à entrées/sorties, réseaux de Petri, …) qui permet de calculer le graphe de tous les changements d'états du système. L'étape de modélisation terminée, le processus de vérification consiste à se donner une propriété que le graphe doit posséder. Cette propriété est exprimée dans une certain logique temporelle (LTL, CTL, μ-calcul, …).
Outre leur efficacité algorithmique, le principal avantage de ces techniques est qu'elles permettent une étude exhaustive du modèle. Mais cette exhaustivité requiert, en général, le stockage de l'ensemble du graphe. Le plus souvent les études menées avec des techniques de vérification se heurtent au problème de l'explosion combinatoire du modèle.
Face à ce problème les algorithmes et les structures de données liés à l'étude du graphe des comportements ont été le sujet de nombreuses recherches. Les solutions apportées peuvent être classées en deux catégories : la réduction du graphe réellement nécessaire à la vérification (techniques dites « à la volée », ordres partiels, …) ou le codage symbolique du graphe (à l'aide de diagrammes de décision binaires ou BDD, d'arbres partagés, …).
Depuis 2003 nous avons doté le langage AltaRica d'outils de vérification de modèles :
Acheck permet de calculer le graphe des comportements du système et des propriétés (sous forme d'ensembles d'états et/ou de transitions) sur ce graphe. Dans cet outil le graphe est codé en extension ce qui borne son utilisation à des systèmes relativement petits (mais la taille nécessaire au stockage est très dépendante du nombre de variables du système).
Mec V[6] possède les mêmes fonctionnalités que **acheck**. Les deux grandes différences entre les outils sont :

1. Le codage du graphe : Mec V code le graphe de manière symbolique en utilisant un paquetage BDD (Diagrammes de Décision Binaires). Ceci permet d'augmenter la taille des systèmes étudiés.
2. La logique de spécification de propriétés : L'outil possède un langage de spécification nettement plus élaboré que **acheck**

qui permet, par exemple, le calcul de point fixes imbriqués sur la structure du graphe.

Parmi les propriétés que ces outils permettent de vérifier nous pouvons citer :

- Les états bloquants (*deadlocks*)
- Les propriétés de sûreté : « est-ce que tel situation peut se produire ? »
- L'inévitabilité : « quels sont les états à partir desquels une situation critique devient inévitable ? »
- Les comportements cycliques particuliers : réinitialisation, vivacité, …
- La bisimulation : « est-ce que deux noeuds sont équivalents ? »

Etude d'un système hydraulique embarqué

Dans le cadre du projet européen ESACS, l'ONERA a présenté l'étude d'un système hydraulique d'un avion proche de l'A320 [5]. Dans cette section nous reprenons une partie de leurs travaux et présentons les résultats obtenus à l'aide des outils développés au LaBRI.

Le système étudié

Le système étudié est un système hydraulique alimentant les éléments permettant le contrôle d'un avion en vol et au sol.
Ce système hydraulique est constitué de trois circuits de distribution quasi-indépendants qui fournissent une puissance hydraulique suffisante aux consommateurs. Pour l'essentiel ces circuits sont constitués de pompes, de vannes, de réservoirs et de tuyaux. Tous ces éléments sont sujets à des défaillances non réparables.
Afin d'assurer une pression suffisante aux consommateurs une architecture de sécurité a été mise en place : certains composants sont en redondance ou ont été équipés de mécanismes de sûreté permettant de palier à une défaillance globale du circuit.
L'objectif de l'étude [5] est de déterminer les scénarios qui mènent à une perte de puissance hydraulique suffisante pour provoquer le dysfonctionnement des commandes.
L'étude de ce système à l'aide de méthodes issues de la SdF se heurte à deux types de difficultés :

1. Des scénarios inexistants : Les coupes mises en évidence par une étude par arbres de défaillances ne sont valables que si les pannes apparaissent dans un ordre donné : la panne p1 suivie de p2 provoquent l'événement redouté alors que ce n'est pas le cas lorsque p2 est suivie de p1.
2. Des phénomènes transitoires : L'événement redouté peut apparaître mais ne pas persister suite à une reconfiguration du système.

Ces difficultés peuvent être résolues si l'on utilise des outils de vérification de modèles. En effet, d'une part ces outils considèrent les comportements en terme de séquences ordonnées d'événements. D'autre part leur langage de spécification de propriétés permet de calculer des propriétés de sûreté plus sophistiquées qui prennent en compte aussi bien ce qui précède l'événement redouté que ce qui suit ; ainsi les phénomènes transitoires peuvent être pris en compte.

Modélisation du système

Nous ne présentons pas ici le modèle du système étudié. Cette modélisation peut être trouvée dans [5]. Toutefois nous pouvons donner quelques mesures de la taille de la description AltaRica :

- Nombre de niveaux hiérarchiques : 3 ;
- Nombre de variables de flux : 149 (booléennes) ;
- Nombre de variables d'état : 31 dont 1 numérique et 30 booléennes ;
- Nombre d'événements : 36 dont 22 pannes élémentaires et 14 actions ;
- Nombre d'assertions : 149 (une par variable de flux).

Propriétés étudiées

Les outils **acheck** et **Mec V** calculent un graphe d'états dont les arcs sont étiquetés par les événements globaux du système ; les états sont des affectations des variables d'états et de flux.
Les propriétés calculées par ces outils sont des ensembles d'états ou de transitions. Ci-dessous est présentée une partie du fichier des spécifications fournies à **acheck** pour cette étude. Nous commentons certaines d'entre elles (encadrées).
La première propriété calculée (*dead*) est définie comme un ensemble d'états : c'est l'ensemble de tous les états (mot-clé *any_s*) duquel sont retirés les états qui sont à l'origine (opérateur *src*) d'une transition qui appartient à l'ensemble *any_t-self_epsilon* (qui est l'ensemble de toutes les transitions duquel ont été retirées les transitions qui ont le même état de départ et d'arrivée et qui sont étiquetées par l'événement *epsilon*). Cette propriété permet de calculer l'ensemble des états qui n'ont pas de successeur par une transition ; en d'autres *dead* est l'ensemble des états bloquants du système.
La propriété *trERPhydNPhyd* est un ensemble de transitions. L'opérateur *trace* permet de calculer un ensemble de transitions qui forment un plus court chemin entre deux ensembles d'états et dont les transitions appartiennent à un ensemble donné. Ici l'ensemble des états de départ est celui des états initiaux, l'ensemble des états d'arrivée est l'ensemble des états qui sont à l'origine d'une transition de l'ensemble *PbERPhydNPhyd* et les transitions du chemin peuvent être quelconques (elles sont spécifiées dans *any_t*). Cette propriété calcule un plus court chemin de l'état initial du système vers l'état redouté.
La propriété *exampleERPhydNPhyd* est un calcul d'accessibilité. L'opérateur *reach* permet de calculer tous les états qui sont accessibles depuis un ensemble d'états de départ par des chemins dont les transitions appartiennent à un ensemble donné. Ici, l'ensemble calculé est celui des états accessibles depuis l'état initial du système par des transitions qui mènent à l'état redouté.

```
with Main do
 // Les configurations bloquantes sont celles sans successeurs.
 dead := any_s - src(any_t - self_epsilon);

 // Les événements qui peuvent être répétés.
 boucle := loop(any_t,any_t);

 // Toutes les pannes possibles.
 failure := label failure;

 // Toutes les actions de contrôle possibles.
 action := label action;

 // Des configurations redoutées liées aux pertes de la puissance
 // hydraulique.
 ERPhydNPhyd := [~Phyd.O & ~NPhyd.O];

 // Des actions répétables qui laissent le système dans des états redoutes
 PbERPhydNPhyd := loop( rsrc(ERPhydNPhyd),
                        rsrc(ERPhydNPhyd) & action);

 // Problèmes temporaires.
 // Des scénarios conduisant aux différents problèmes
 trERPhydNPhyd := trace( initial,
                         any_t,
                         src(PbERPhydNPhyd));

 // les actions qui permettent de sortir des situations critiques.
 goodActionERPhydERNPhyd :=  action &
                             rsrc(ERPhydNPhyd) &
                             rtgt(any_s - ERPhydNPhyd);

 // Des scénarios critiques et les décisions a prendre
 exampleERPhydNPhyd := reach( initial,
                              trERPhydNPhyd |
                              PbERPhydNPhyd |
                              goodActionERPhydERNPhyd);

 // Sauvegarde des comportements critiques pour visualisation
 dot(exampleERPhydNPhyd, trERPhydNPhyd |
                         PbERPhydNPhyd |
                         goodActionERPhydERNPhyd)
     > '$NODENAME.ERPhydNPhyd.dot';

 // Problèmes persistants
 // Les états n'offrant pas de bonnes actions
 PersERPhydNPhyd :=
       ERPhydNPhyd &
       (src(action - goodActionERPhydERNPhyd)
       -src(goodActionERPhydERNPhyd));

 // Des actions répétables qui laissent le système dans des états redoutes
 PbPersERPhydNPhyd :=
    loop(rsrc(PersERPhydNPhyd),
         rsrc(PersERPhydNPhyd) & action);

 // Des scénarios conduisant aux différents problèmes
 trPbPersERPhydNPhyd := trace(initial,
                              any_t,
                              src(PbPersERPhydNPhyd));

 // Des scénarios très critiques
```

```
examplePersERPhydNPhyd := reach( initial,
                                 trPbPersERPhydNPhyd |
                                 PbPersERPhydNPhyd);

// Sauvegarde des comportements très critiques pour visualisation
dot(examplePersERPhydNPhyd,
    trPbPersERPhydNPhyd |
    PbPersERPhydNPhyd)
    > '$NODENAME.PersERPhydNPhyd.dot';

// Sauvegarde des résultats dans des fichiers
show(all)                    > '$NODENAME.prop';
test(dead,0)                 > '$NODENAME.res';
test(ERPhydNPhyd,0)          >> '$NODENAME.res';
test(PbPersERPhydNPhyd,0)    >> '$NODENAME.res';
done
```

Ci-dessous sont présentées les spécifications utilisées pour **Mec V**. Les propriétés calculées sont analogues, toutefois la façon de les spécifier est différente. En effet **Mec V** manipule de relation *n*-aire sur des ensembles d'états et/ou de transitions ; dès lors les propriétés calculées par cet outil sont exprimées en terme d'opérations ensemblistes. **Mec V** va plus loin en proposant les opérateurs du μ-calcul modal[6].
Par exemple la propriété *Rs(t)* est la relation unaire (c.-à-d. un ensemble) qui contient tous les états accessibles depuis l'état initial. La définition de *Rs(t)* est ni plus ni moins que son expression habituelle en terme de point fixe : un état *t* est accessible si et seulement si :

1. C'est un état initial (*Main!init(t)*) ou (|) ;
2. Il existe un état *s* et un événement *e* tels que
   a. *s* est un accessible (*Rs(s)*) et (**&**)
   b. il existe une transition de *s* à *t* étiquetée *e* (*Main!t(s,e,t)*).

```
:ar-display Main
:rel-cardinal Main!init
:rel-cardinal Main!t

// calcul des états accessibles
Rs(t) += Main!init(t) | <s><e>(Rs(s) & Main!t(s,e,t));
:rel-cardinal Rs

// calcul des états redoutés
ERPhydNPhyd(s) := Rs(s) & ~s.Phyd_O & ~s.NPhyd_O;
:rel-cardinal ERPhydNPhyd

// relation de transition accessible
Rt(s,e,t) := Rs(s) & Rs(t) & Main!t(s,e,t);
:rel-cardinal Rt

// Les transitions de panne
Failure(s,e,t) := Rt(s,e,t) &
                  (t.nbFailures != s.nbFailures);
:rel-cardinal Failure

// Les transitions qui ne sont pas des pannes
Action(s,e,t) := Rt(s,e,t) &
                 (t.nbFailures = s.nbFailures)&
                 (s != t);
:rel-cardinal Action

// Actions qui sortent d'un état redouté
GoodAction(s,e,t) := Action(s,e,t) &
                     ERPhydNPhyd(s) &
                     ~ERPhydNPhyd(t);
:rel-cardinal GoodAction

// Les états redoutés desquels on ne peut atteindre qu'un autre état redouté
InevERPhydNPhyd(s) -=
        ERPhydNPhyd(s) &
        [e][t] (Rt(s,e,t) => InevERPhydNPhyd(t));
:rel-cardinal InevERPhydNPhyd
```

Résultats et bilan

Nous avons calculé les propriétés précédentes en ajoutant au modèle un compteur de pannes que nous avons restreint afin de maîtriser l'explosion combinatoire. Nous avons étudié les scénarios critiques contenant jusqu'à 4 pannes.

Le tableau suivant synthétise les résultats des calculs effectués avec les outils. *ER* correspond à la propriété *ERPhydNPhyd* calculée par **acheck** et **Mec V**, *PbER* correspond à *PbERPhydNPhyd* et *PersER* correspond à *PbPersERPhydNPhyd*.

Les ensembles ER, PbER et PersER représentent respectivement les configurations redoutées et des ensembles de transitions y conduisant.

Propriété	Nombre de pannes				
	0	1	2	3	4
# états	64	1.472	16.256	114.816	582.976
# trans	544	13.920	169.152	1.302.592	7.154.592
ER	0	0	262	6.327	68.694
PbER	0	0	1390	37.006	
PersER	0	0	0	724	12.352

Remarques :

1. Ce tableau fait apparaître l'accroissement exponentiel de la taille du modèle en fonction du nombre de pannes permises.
2. L'événement redouté est impossible pour 0 ou 1 panne, devient possible avec 2 pannes tout en restant réparable et est définitif à partir de 3 pannes. Ce résultat confirme ceux parus dans [5].

Les tables suivantes présentent les performances respectives de chaque outil en terme de temps de calcul et de consommation mémoire. Les tests ont été réalisés sur un processeur Xeon cadencé à 2.2Ghz et disposant de 6Go de mémoire.

Temps CPU	Nombre de pannes				
	0	1	2	3	4
Acheck	0.4s	8s	1m36s	22m32s	?
Mec V	1m47s	6m13s	16m52s	17m24s	68m11s

Mémoire (Mo)	Nombre de pannes				
	0	1	2	3	4
Acheck	15	16	35	227	?
Mec V	127	162	178	312	692

Nous retrouvons ici des résultats déjà connus :

1. **Acheck** demande un temps de calcul et un espace mémoire proportionnels à la taille du graphe construit. Sur cet exemple 4 pannes vont nécessiter un temps de calcul important.
2. **Mec V** est peu efficace sur des petits modèles mais son codage symbolique du graphe lui permet de traiter un plus grand nombre de pannes.

Conclusion

Nous avons montré qu'au-delà des aspects purement fonctionnels, la vérification de modèles pouvait être utilisée avec avantage pour l'étude des dysfonctionnements de systèmes complexes. Si le problème de l'explosion combinatoire des modèles est incontournable, l'amélioration des l'algorithmes implémentés dans les outils devrait permettre de repousser les limites de traitement des modèles étudiés ici.
Les outils **acheck** et **MecV** ont été développés au LaBRI et sont accessibles sur le site du projet AltaRica (http://altarica.labri.fr/).

Remerciements

Cette étude a réalisée dans le cadre du projet européen ESACS (Enhanced Safety Assessment for Complex Systems) et financée par le CERT/ONERA.

Références

[1] A. Arnold, A. Griffault, G. Point et A. Rauzy, The AltaRica formalism for describing concurrent systems, Fundamenta Informaticae, 40:109-124, 2000.
[2] A. Griffault, S. Lajeunesse, G. Point, A. Rauzy, J.-P. Signoret et P. Thomas, Le langage AltaRica, Actes du 11ème congrès λμ'11, 1999.
[3] G. Point, AltaRica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement, Thèse de Doctorat, Université Bordeaux 1, 2000.
[4] G. Point et A. Rauzy, AltaRica – Constraint automata as a description language. European Journal on Automation, 1999.
[5] C. Seguin et C. Kehren, Evaluation qualitative de systèmes physiques pour la sûreté de fonctionnement, Dans les actes de Formalisation des Activités Concurrentes, FAC'03, mars 2003.
[6] A. Vincent, Conception et réalisation d'un vérificateur de modèles AltaRica, Thèse de Doctorat, Université Bordeaux 1, 2003.