

Vérification de modèles AltaRica

Alain Griffault

Aymeric Vincent

LaBRI, Université Bordeaux 1, 351 cours de la Libération, 33400 Talence

Prenom.Nom@labri.u-bordeaux.fr

Résumé : Nous présentons dans cet article les particularités du model-checker que nous avons réalisé : Mec V. Cet outil permet de traiter des modèles au format AltaRica et se caractérise par le grand pouvoir d'expression de la logique utilisée : du μ -calcul sur des relations avec quantificateurs du premier ordre et égalité.

Mots-clés : Vérification et validation de modèles

1 Introduction

AltaRica [Poi00] est un formalisme de description de systèmes. Sa conception a été faite au LaBRI, en partenariat étroit avec des industriels. Deux volontés étaient mêlées au projet dès son lancement en 1997 : avoir un langage de description utilisable facilement, et avoir une sémantique formelle sans ambiguïté.

Initialement, l'outillage prévu autour du langage AltaRica comprenait un *Atelier* pour aider l'utilisateur à la saisie des modèles, et divers compilateurs vers des outils d'analyse. Pour les études de sûreté de fonctionnement, où les résultats sont principalement de nature quantitative, cette approche *compilation* vers d'autres formalismes a donné de bons résultats. Pour les analyses de type *model-checking* pour lesquelles l'utilisateur doit exprimer des propriétés sur le modèle, il s'est avéré que le passage par un traducteur rendait l'expression des propriétés très difficile. C'est pourquoi nous avons décidé de développer un vérificateur de modèle pour le langage AltaRica. Néanmoins, nous avons voulu que cet outil soit le plus général possible, afin qu'il soit facilement adaptable à d'autres formalismes.

Après avoir présenté le formalisme et le langage AltaRica, nous décrirons le langage de spécification proposé par Mec V.

Ce langage a l'expressivité du μ -calcul sur des relations augmenté des quantificateurs du premier ordre et de l'égalité, ce qui le rend beaucoup plus expressif que les logiques temporelles utilisées habituellement en vérification, comme LTL, CTL, ou CTL* [SBB⁺99].

Nous terminerons par un exemple où nous

définissons sans difficulté une relation de bisimulation entre deux nœuds AltaRica, *directement* dans cette logique.

2 AltaRica

La base sous-jacente des modèles AltaRica (en lesquels tout modèle AltaRica se traduit) est constituée par les automates à contraintes.

Pour l'aspect expression, des notions importantes de hiérarchie et donc de synchronisation ont été introduites dans le langage afin de faciliter l'écriture des modèles.

2.1 Les automates à contraintes

Les automates à contraintes décrits par exemple dans [FP94] sont constitués d'états et d'événements qui sont des *instances* de contraintes sur des variables entières, c'est-à-dire des ensembles de valuations de ces variables. Ces contraintes permettent de représenter de manière finie des ensembles infinis d'états et d'événements.

Les outils qui manipulent des modèles AltaRica limitent les valeurs des variables à des domaines finis. L'avantage principal que nous retenons des automates à contraintes est la facilité d'expression. Un état n'est plus un nom, mais une valuation de variables.

La façon de décrire un modèle consiste à énumérer ses transitions, qui sont de la forme (g, e, σ) , où g est une contrainte qui garde la transition, e est un nom d'événement, et σ est une affectation des variables qui s'opère à la suite du déclenchement de la transition.

Par exemple, l'état d'un interrupteur peut être représenté par une variable booléenne qui code le fait qu'il soit ouvert ou fermé. Si l'on modélise par un événement *appui* le fait d'appuyer sur l'interrupteur, le système a deux transitions à contraintes :

(ouvert = vrai, appui, ouvert := faux)
(ouvert = faux, appui, ouvert := vrai)

Ce qui peut s'écrire avec la syntaxe concrète AltaRica de manière concise :

```

node Interrupteur
  state ouvert : bool;
  event appui;
  trans
    true |- appui -> ouvert := ~ouvert;
edon

```

D'autre part, on peut introduire une contrainte globale que l'on appelle *assertion*. Par exemple, on peut modéliser le fait que l'interrupteur soit à deux vitesses (lorsque l'événement *changeVitesse* survient), et qu'il soit impossible d'être en vitesse *rapide* si l'interrupteur n'est pas fermé. Voici une écriture possible de cet interrupteur en AltaRica :

```

node InterrupteurDeuxVitesse
  state
    ouvert : bool;
    rapide : bool;
  event appui, changeVitesse;
  trans
    true |- appui -> ouvert := ~ouvert;
    true |- changeVitesse ->
      rapide := ~rapide;
  assert
    rapide => ~ouvert;
  init
    ouvert := false, rapide := false;
edon

```

Notez dans cet exemple l'ajout d'une information supplémentaire (à destination d'outils comme un simulateur ou un model-checker) sur les états initiaux : l'interrupteur est initialement en état de fonctionnement dans une vitesse lente.

2.2 La hiérarchie

Nous avons vu comment modéliser le comportement d'un système simple. Mais afin de pouvoir réutiliser des composants et de pouvoir organiser le travail de modélisation, le langage AltaRica permet d'utiliser un nœud AltaRica à l'intérieur d'un autre nœud.

Par exemple, on peut rassembler deux interrupteurs dans un même modèle :

```

node DeuxInterrupteurs
  sub
    I1 : Interrupteur;
    I2 : Interrupteur;
  edon

```

Les états de chacun de ces deux interrupteurs pourront varier de manière totalement asynchrone.

Il y a deux manières complémentaires de décrire l'interaction entre plusieurs sous-nœuds. La première consiste à introduire une notion de visibilité sur certaines variables, afin de pouvoir lier les contraintes portant sur des variables de sous-composants. La deuxième consiste à *synchroniser* certains événements, pour imposer qu'ils aient lieu au même instant. Intuitivement, elles correspondent aux classiques distinctions entre les communications par variables partagées et celles par messages.

2.3 Les variables de flux

Le langage propose en plus des variables d'état une deuxième classe de variables, qui peuvent prendre leurs valeurs sur les mêmes domaines. Ces variables dites *de flux* sont uniquement contraintes par l'assertion du nœud dans lequel elles apparaissent. On ne peut pas directement leur affecter de valeur lors du franchissement d'une transition.

Ajoutons à notre exemple d'interrupteur deux flux symbolisant les deux bornes de courant. On choisit par simplicité de les coder par des booléens, vrai et faux symbolisant chacun une tension différente.

```

node Interrupteur
  state ouvert : bool;
  flow f1, f2 : bool;
  event appui;
  trans
    true |- appui -> ouvert := ~ouvert;
  assert
    (~ouvert) => (f1 = f2);
edon

```

L'assertion traduit bien l'idée physique que les deux tensions sont les mêmes si l'interrupteur est fermé.

L'intérêt des variables de flux est qu'elles constituent la partie visible du nœud et sont donc exportées vers le nœud parent.

Modélisons le va-et-vient de la figure 1 avec deux interrupteurs "va-et-vient".

```

node InterrupteurVaEtVient
  state choix : [1, 2];
  flow f, f1, f2 : bool;
  event appui;
  trans
    choix = 1 |- appui -> choix := 2;
    choix = 2 |- appui -> choix := 1;
  assert
    (choix = 1) => (f = f1);
    (choix = 2) => (f = f2);
edon

```

```

node VaEtVient
  flow f1, f2 : bool;
  sub I1, I2 : InterrupteurVaEtVient;
  assert
    f1 = I1.f;
    I1.f1 = I2.f1 & I1.f2 = I2.f2;
    f2 = I2.f;
edon

```

Notez que l’assertion du nœud *VaEtVient* correspond dans ce cas à l’idée intuitive de relier les bornes avec des fils conducteurs, comme on peut le voir sur la figure 1.

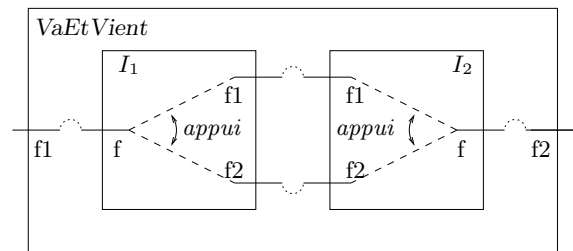


FIG. 1 – Un va et vient électrique

2.4 Les vecteurs d’événements

On peut décider de synchroniser des événements appartenant à des nœuds différents. Cela signifie qu’ils ne pourront plus avoir lieu que en même temps, ensemble. Pour cela, on introduit la notion de *vecteur d’événements*. Les vecteurs d’événements sont en fait les véritables représentants des actions que peut effectuer un système.

Il est bon ici de préciser une particularité du modèle AltaRica : tous les composants sont dotés d’un événement particulier “ ε ” qui peut toujours être exécuté et qui conserve les valeurs des variables d’états.

Ainsi, les événements d’un sous-nœud qui ne sont pas explicitement synchronisés avec d’autres le sont implicitement avec les événements “ ε ” du nœud parent et des autres sous-nœuds du nœud parent. Ceci explique la sémantique asynchrone à laquelle il était fait allusion plus haut.

Ajoutons à notre va-et-vient un événement *appui*, qui correspondra à l’appui sur l’un ou l’autre des deux interrupteurs le composant.

```

node VaEtVient
  flow f1, f2 : bool;
  sub I1, I2 : InterrupteurVaEtVient;
  event appui
  trans
    true |- appui -> ;

```

```

sync
  <appui, I1.appui>;
  <appui, I2.appui>;
assert
  f1 = I1.f;
  I1.f1 = I2.f1 & I1.f2 = I2.f2;
  f2 = I2.f;
edon

```

Notez qu’il n’est pas nécessaire de déclarer une variable d’état pour pouvoir décrire des transitions. Intuitivement, un nœud père joue le rôle d’un contrôleur pour ses fils. Ici le contrôleur est simplement sans état.

2.5 Les priorités

Il est possible de spécifier un ordre partiel strict sur les événements d’un nœud pour modéliser leur priorité relative.

Si l’on écrit par exemple

```
event appui < panne
```

cela déclare deux événements *appui* et *panne* tels que les transitions *appui* ne peuvent être franchies que si aucune transition *panne* n’est franchissable.

2.6 La diffusion

Il peut arriver qu’un émetteur doive se synchroniser avec plusieurs récepteurs. Cependant, si les modèles des récepteurs prennent en compte par exemple des états de panne dans lesquels l’événement de réception ne peut plus avoir lieu, on ne peut plus imposer que tous les récepteurs se synchronisent avec l’émetteur sans bloquer tous les autres protagonistes dès que l’un seulement des récepteurs est en panne.

Pour modéliser aisément ce cas courant, AltaRica introduit les vecteurs de diffusion, qui sont des vecteurs de synchronisation dans lesquels certains événements sont optionnels. La sémantique est la suivante : tout événement exécutable apparaissant dans le vecteur de diffusion sera exécuté si ce vecteur l’est. Les événements optionnels ne peuvent empêcher l’exécution du vecteur.

Si par exemple on imagine que le nœud *TeleLampe (TL)* modélise une lampe commandée par radio qui peut tomber en panne, et que le nœud *TeleRupteur (TR)* permet de commander plusieurs de ces lampes, alors la demande par l’interrupteur d’extinction de trois lampes pourra s’écrire :

```
sync <TR.off, TL1.off?, TL2.off?, TL3.off?>
```

La diffusion introduit de manière transparente une priorité sur les *vecteurs d’événements* qui

sont déduits du vecteur de diffusion. Le vecteur précédent est *équivalent* à la déclaration suivante :

```
event off3 > off2 > off1 > off0;
sync
  <off3, TR.off, TL1.off, TL2.off, TL3.off>;
  <off2, TR.off, TL1.off, TL2.off>;
  <off2, TR.off, TL1.off, TL3.off>;
  <off2, TR.off, TL2.off, TL3.off>;
  <off1, TR.off, TL1.off>;
  <off1, TR.off, TL2.off>;
  <off1, TR.off, TL3.off>;
  <off0, TR.off>;
```

3 Aspect outil

Mec V, comme les autres outils actuellement disponibles pour exploiter des modèles AltaRica, procède en plusieurs étapes afin de calculer la relation de transition d'un nœud donné :

1. Incorporation des variables des sous-nœuds dans le nœud courant.
2. Calcul de l'assertion du nœud en tenant compte des assertions des sous-nœuds.
3. Calcul de l'état initial en tenant compte des sous-nœuds.
4. Calcul de la relation de transition de chacune des transitions locales du nœud.
5. Calcul de la relation de transition de chacun des événements locaux.
6. Calcul de l'événement ε local du nœud.
7. Calcul des vecteurs d'événements apparaissant dans le nœud ainsi que de leur relation de transition.
8. Résolution des contraintes de priorité.

Cette façon de procéder respecte la sémantique du langage AltaRica, comme cela est décrit dans [AGPR00] et [Poi00].

Comme nous ne considérons que des modèles finis, nous utilisons des diagrammes de décision binaires (BDDs) [Bry86] avec arcs négatifs, ainsi que des vecteurs de BDDs pour les calculs arithmétiques sur les domaines finis.

Le choix des BDDs comme structure de données nous permet d'avoir un langage de spécification très expressif.

4 Le langage de spécification

Nous allons maintenant présenter le langage proposé par Mec V pour spécifier des propriétés sur des modèles AltaRica.

Le choix que nous avons fait est de proposer un moteur de calcul très expressif. Il intègre la définition de relations comme solutions de systèmes d'équations de points fixes avec un nombre quelconque d'alternances, tels que décrits dans [Mad97] et [AN01], et inspirés de la syntaxe de l'outil Toupie [CR93].

Grâce aux BDDs, le calcul de la projection et de la cylindrification sont aisés et efficaces [Bry86]. Le langage de spécification reflète cette possibilité et offre donc les quantificateurs du premier ordre. On pourra se référer à [AG94] pour voir que les quantificateurs du premier ordre et les opérateurs de points fixes sont deux façons orthogonales et non équivalentes de spécifier. L'égalité entre variables du premier ordre est aussi exprimable dans le langage.

Enfin, un mécanisme d'inférence de types permet d'alléger l'écriture des expressions.

4.1 Atomes

Les formules de base permettent d'exprimer des propriétés sur les variables. Ces variables peuvent prendre leur valeur dans les mêmes domaines finis que les variables AltaRica :

- booléens (**bool**)
- intervalle fini d'entiers ($[1, 10]$)
- énumération (**{ouvert, ferme}**)

ou bien sur deux domaines supplémentaires représentant les configurations d'un nœud et les vecteurs d'événements d'un nœud.

- configuration (**Interrupteur!c**)
- vecteur d'événements (**Interrupteur!ev**)

Les opérateurs booléens classiques de conjonction (&), de disjonction (|), de négation (~) sont dans le langage, ainsi que les opérateurs arithmétiques usuels (+, -), les opérateurs de comparaison (<, <=, >, >=, =); tous ne s'appliquant bien sûr pas à n'importe quel type.

Les variables d'état ou de flux des nœuds et de leurs sous-nœuds sont accessibles par notation pointée. Par exemple, si **c** est de type **VaEtVient!c**, alors on peut écrire une formule comme : **c.I1.f & c.f1**.

Pour les vecteurs d'événements, de manière similaire, si **e** est de type **VaEtVient!ev**, on peut écrire **e. = appui** ou **e.I1. = appui**. Notez que sans le **.** qui suit le nom du vecteur, c'est tout le vecteur qui est référencé et pas seulement l'événement local. On peut par exemple écrire **e.I1 = e.I2** pour choisir les **e** dont les composantes pour chacun des sous-nœuds **I1** et **I2** sont les mêmes.

De plus, il est possible d'utiliser (et nous verrons plus loin, de définir) des relations. Pour chaque

nœud AltaRica A , deux prédicats sont définis :

- $A!init$ de type $A!c$ qui représente les configurations initiales du nœud A
- $A!t$ de type $A!c \times A!ev \times A!c$ qui représente la relation de transition du nœud A

Par exemple, si c est de type $VaEtVient!c$, on peut écrire : $VaEtVient!init(c) \ \& \ c.f1$

4.2 Quantificateurs

Il est possible d'introduire dans les expressions des variables quantifiées. La syntaxe choisie est inspirée des modalités existentielle et universelle classiques en logique modale. Voici la syntaxe concrète pour chacun des deux quantificateurs.

$$\begin{array}{ll} \forall x \in D, \phi(x) & [x : D] \ \text{phi}(x) \\ \exists x \in D, \phi(x) & \langle x : D \rangle \ \text{phi}(x) \end{array}$$

Par exemple, si on veut exprimer la propriété (fausse) que tous les successeurs immédiats par appui des états initiaux du nœud $VaEtVient$ ont leur flux $f1$ à vrai :

```
[c'] ((<c><e>
(VaEtVient!init(c) & e. = appui &
VaEtVient!t(c,e,c')) => c'.f1);
```

4.3 Systèmes d'équations

Afin de pouvoir exprimer des propriétés globales du modèle, i.e. sur ses comportements, il est possible d'exprimer des calculs de points fixes.

Voici sur un exemple la syntaxe du calcul d'une relation par (plus petit) point fixe ; il s'agit de la relation décrivant les configurations accessibles d'un nœud $VaEtVient$.

```
AccVV(t) += VaEtVient!init(t) |
<s><e> (VaEtVient!t(s,e,t) & AccVV(s));
```

Afin de pouvoir définir des points fixes alternants (i.e. des relations interdépendantes qui ne sont pas toutes des plus petits ou toutes des plus grands points fixes), on introduit la notion de systèmes d'équations de points fixes.

Un système est délimité par les deux mots-clés **begin** et **end**, et l'imbrication des différentes relations est spécifiée par un entier encapsulé dans le symbole de point fixe.

Voici un exemple adapté de la bibliothèque d'exemples de Toupie [CR93].

```
begin
  tau(x : A!c) -1= aux(x);
  local
    aux(x) +2=
```

```
<y><e> (A!t(x,e,y) & aux(y)) |
<y><e> (A!t(x,e,y) & P(y) & tau(y));
```

end

Ce calcul définit la relation tau comme les configurations qui appartiennent à des boucles infinies passant infiniment souvent par une configuration satisfaisant P .

5 Exemple

Voici un exemple plus conséquent et qui montre toute la puissance et l'intérêt de Mec V : il est possible d'exprimer directement et simplement dans notre logique la relation de bisimulation entre configurations de deux nœuds AltaRica différents.

Nous allons calculer la relation de bisimulation entre le nœud *Interrupteur* et le nœud *VaEtVient*, afin de montrer qu'ils sont bisimilaires.

Ce calcul est utile car bien que *VaEtVient* ait 16 états et 192 transitions, *Interrupteur* n'a que 6 états et 36 transitions. La sémantique AltaRica nous assure que pour tout nœud dans lequel apparaît un *VaEtVient*, on peut remplacer le *VaEtVient* par un *Interrupteur* tout en conservant des modèles bisimilaires [AGPR00].

Une relation de bisimulation exprime une équivalence entre deux états d'un système en se basant uniquement sur la partie visible de ce système. Dans le cas usuel où le système est constitué de deux automates possédant des transitions étiquetées, la relation de bisimulation ne prend en compte que les étiquettes des transitions [Mil80].

Ici, conformément à [AGPR00], nous devons définir deux relations d'équivalence : la classique entre les événements, et une seconde entre les configurations puisque les variables de flux sont une partie visible du système.

```
eqEv(a : Interrupteur!ev,
     b : VaEtVient!ev) :=
(a. = appui & b. = appui) |
(a. = "" & b. = "");
```

```
eqC(a : Interrupteur!c, b : VaEtVient!c) :=
(a.f1 = b.f1) & (a.f2 = b.f2);
```

Enfin, il ne reste plus qu'à écrire la relation par plus grand point fixe définissant la relation de bisimulation.

```
bisim(a, a') -=
eqC(a, a') &
([e][s] (Interrupteur!t(a,e,s) =>
<e'><s'> (VaEtVient!t(a',e',s') &
```

```

    eqEv(e,e') & bisim(s,s')))) &
([e'] [s'] (VaEtVient!t(a',e',s') =>
<e><s> (Interrupteur!t(a,e,s) &
    eqEv(e,e') & bisim(s,s'))));

```

A l'heure actuelle, Mec V ne permet que d'afficher une relation, on est donc obligé pour vérifier que les deux systèmes sont bisimilaires de définir une relation toute simple sur un booléen qui vaudra vrai si les systèmes sont bisimilaires. En AltaRica, cette relation porte sur toutes les configurations, car elles sont toutes potentiellement initiales.

```

estBisim(x : bool) :=
  x = ((([a]<a'> bisim(a,a')) &
    ([a']<a> bisim(a,a'))));

```

Lorsque l'on demande à Mec V d'afficher la relation *estBisim*, on voit qu'elle impose que *x* vaille vrai :

```

[mec5] :display estBisim
(true)
[mec5]

```

6 Conclusion

Les quelques expérimentations que nous avons faites avec Mec V sont prometteuses et répondent bien à notre attente d'un model-checker très expressif autorisant ce que les BDDs nous permettent d'exprimer.

Les perspectives offertes sont vastes, car bien que le choix de cette expressivité soit principalement dû à la structure de données employée, il y a fort à parier que les exemples que nous traiterons nous encourageront à ajouter à Mec V d'autres structures de données, ou même d'autres techniques de vérification.

Nous envisageons aussi d'utiliser Mec V dans le domaine très actif de la synthèse de contrôleurs pour implémenter la méthode décrite dans [Vin01] et étendue dans [AVW03].

Références

- [AG94] Miklos Ajtai and Yuri Gurevich. Data-log vs. first-order logic. In *Proceedings of the 30th IEEE symposium on Foundations of computer science*, pages 562–588. Academic Press, Inc., 1994.
- [AGPR00] André Arnold, Alain Griffault, Gérald Point, and Antoine Rauzy. The alta-rica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40 :109–124, 2000.
- [AN01] André Arnold and Damian Niwiński. *Rudiments of μ -calculus*. Studies in Logic and the Foundations of Mathematics. North-Holland, 2001.
- [AVW03] André Arnold, Aymeric Vincent, and Igor Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, 303(1) :7–34, June 2003.
- [Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8) :677–691, August 1986.
- [CR93] Marc-Michel Corsini and Antoine Rauzy. Toupie user's manual. Research Report 586-93, LaBRI, 1993.
- [FP94] Laurent Fribourg and Marcos Veloso Peixoto. Automates concurrents à contraintes. *Technique et Science Informatiques*, 13(6) :837–866, 1994.
- [Mad97] Angelika Mader. *Verification of modal properties using boolean equation systems*. PhD thesis, Fakultät Informatik, Technische Universität München, 1997.
- [Mil80] Robin Milner. *A calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1980.
- [Poi00] Gérald Point. *AltaRica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement*. PhD thesis, LaBRI, Université Bordeaux I, January 2000.
- [SBB⁺99] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciens : Techniques et outils du model-checking*. Génie Logiciel. Vuibert Informatique, 1999.
- [Vin01] Aymeric Vincent. Synthèse de contrôleurs et stratégies gagnantes dans les jeux de parité. In *MSR'01*, 2001.