Verimag

Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Rapport de stage : Étude des possibilités d'utilisation de Aspic dans Ac2lus

GUFFON Florian

20 août 2010

UNIVERSITE JOSEPH FOURIER SCIENCES TECHNOLOGIE MEDECINE

CNRS

Grenoble INP

Table des matières

1 Introduction **3**
1.1 Contexte . . . 3
1.2 Problématique . . . 3

2 Prérequis **3**
2.1 LUSTRE . . . 3
2.1.1 Introduction . . . 3
2.1.2 Compilation . . . 4
2.2 Ac2lus . . . 4
2.2.1 Introduction . . . 4
2.2.2 Obtention des courbes d'arrivées . . . 6
2.3 Outils . . . 7
2.3.1 Kind . . . 8
2.3.2 Nbac . . . 8
2.3.3 Aspic . . . 8

3 Vers l'utilisation de aspic dans AC2lus **9**
3.1 Expérimentation aspic . . . 9
3.1.1 De LUSTRE vers ASPIC . . . 9
3.1.2 Exemple . . . 9
3.2 Expérimentation - AC2LUS . . . 12
3.3 Problèmes rencontrés . . . 13

4 Conclusion **13**
4.1 Résumé . . . 13
4.2 Perspectives . . . 13

5 Annexes **14**
5.1 Annexe A : Résultat de l'exemple Aspic . . . 14
5.2 Annexe B : Automate issue de aspic du noeud VersCent . . . 15
5.3 Annexe C : Fichier binary_search généré par ac2lus . . . 15

1 Introduction

1.1 Contexte

Ce stage d'une durée de cinq semaines s'est déroulé au sein du laboratoire Verimag[1], spécialisé dans la recherche informatique et les systèmes embarqués, il se compose de trois équipes : Tempo, DCS et l'équipe synchrone dans laquelle j'ai effectué mon stage, qui a développé le langage synchrone Lustre (2.1) et travaille principalement sur la vérification et preuve sur des programmes via le développement de plusieurs outils ayant comme socle le langage Lustre. On comprend l'importance de ces programmes de vérification dans les systèmes embarqués tels que ceux utilisés dans les transports ou il faut s'assurer qu'un événement déclenché par le pilote (par exemple : la sortie du train d'atterrissage) s'effectue correctement d'un point du vue informatique et dans un intervalle temps défini. C'est dans ce contexte que l'outil Ac2lus (2.2) a été développé. C'est un outil permettant de calculer des propriétés (sur le nombre d'événements) en sortie d'un système modéliser en Lustre en fonction de ses entrées et du temps qui, comme nous le verrons, s'appuie sur des outils preuve de propriétés.

1.2 Problématique

L'objet de mon stage est d'étudier l'utilisation de Aspic [2] (Aspic est un outil permettant de prouver des propriétés concernant des programmes et renvoie des invariants) dans le cadre d'une future utilisation dans Ac2lus. Nous pensons que Aspic pourrait remplacer les outils de preuve utilisés dans Ac2lus et en améliorer les performances. Aspic, utilise différentes méthodes d'accélération afin de prouver des propriétés et de renvoyer des invariants sur le programme testé. L'objectif est d'étudier comment peut être utilisé Aspic à partir d'un programme lustre, notamment au niveau de la compatibilité entre Lustre et le format d'entrée Aspic, puis, si des résultats satisfaisant sont obtenues, d'étudier comment peut être intégré Aspic dans Ac2lus.

Plan : Dans une première partie, nous présenterons les prérequis 2 nécessaires :
- Le langage Lustre 2.1 et sa compilation 2.1.2
- Le programme Ac2lus 2.2 : son fonctionnement et la manière dont on obtient les courbes de sorties 2.2.2 ainsi que les outils utilisés 2.3 : kind, Nbac et aspic.

Puis, nous regarderons comment fonctionne aspic et comparerons les résultats obtenus selon l'outil 3.1 à l'aide d'exemples 3.1.2 et tenterons de tracer l'évolution de l'automate 3.1.2 tout au long de la chaîne Lustre vers aspic.
Nous étudierons ensuite comment aspic pourrait être utilisé dans Ac2lus 3.2.
La partie suivante concerne les problèmes rencontrés lors de ces expérimentations et qui ont freiné notre progression.3.3
Enfin, nous reviendrons brièvement sur le travail réalisé lors de la conclusion 4 et donnerons les perspectives.

2 Prérequis

2.1 LUSTRE

pour plus d'informations voir : A turorial of lustre [6]

2.1.1 Introduction

Lustre est un langage synchrone développé par le laboratoire Verimag depuis plusieurs années (plus de 20 ans), il est principalement utilisé dans le "contrôle des systèmes critiques", il est le socle de l'outil SCADE (Esterel Technologies) utilisé par AIRBUS,schneider electric...
A l'origine, Lustre était un langage conçu pour l'implémentation, mais nous en nous servons à présent également pour la modélisation. Un programme Lustre est constitué de noeuds, un noeud à pour entrées et sorties des flux de valeurs et est synchrone avec les autres noeuds du programme selon une horloge

en temps discret. Un noeud est exécuté à chaque tic horloge, les variables (y compris les entrés/sorties) peuvent avoir des valeurs différentes à chaque tic horloge.

```
node incr(init:int;I,reset:bool) returns (result:int)
let
        result=init ->  if reset then init
                        else if I then pre(result)+1
                             else pre(result);
tel;
```

On note que :
- la valeur située à gauche de l'opérateur -> correspond à l'initialisation de la variable.
- l'opérateur pre est la *mémoire* de LUSTRE, pre(X) correspond à la valeur précédente de X, soit la valeur de la variable X au tic horloge précédent.

A la première exécution de ce noeud, *result* prendra donc la valeur de *init*, puis à chaque tic horloge, si *reset* est *vraie* alors on réinitialise *result* avec la valeur courante de *init* sinon, si I est *vraie* alors on incrémente *result*.

2.1.2 Compilation

La compilation d'un programme Lustre comporte 2 étapes :

1. *lustre fichier.lus MainNode* ou MainNode correspond au noeud à compiler contenue dans fichier.lus. Il est possible que ce fichier contienne d'autres noeuds et que MainNode contienne des appels à ceux-ci, MainNode est le noeud principal. Cette étape ne génère pas d'exécutable mais deux fichiers :
   - MainNode.ec : il contient du code LUSTRE, mais un unique noeud, si MainNode contenait des appels à d'autres noeuds, ils ont été remplacé par les noeuds en question.
   - MainNode.oc : le format .oc (object code) a été conçu pour représenter les automates générés par la compilation d'un programme LUSTRE.
2. *lux MainNode.oc* : cette étape génère du code C à partir du fichier MainNode.ec qui engendrera un exécutable nommé MainNode.

On peut également simuler un noeud dans une interface graphique grâce à la commande :
luciole fichier.lus MainNode
le simulateur ouvre une fenêtre ou sont représentées les entrés et sorties du noeud MainNode, on peut alors exécuter le programme, soit pas par pas (ou l'exécution de chaque pas est déclenché par un clic sur le bouton step), soit en temps réel. On peut alors modifier les entrés en temps réel et observer les conséquences sur les sorties.

2.2 Ac2lus

pour plus d'informations sur ac2lus, lire le rapport complet [5]

2.2.1 Introduction

ac2lus est un programme permettant d'analyser les performances en temps d'un système, et de prévoir, en fonction d'un système et de ses entrées, des propriétés sur les sorties de celui ci. Les propriétés renvoyées en sorties sont des propriétés sur le nombre d'événements traités par le programme en fonction d'un intervalle de temps.
Décrivons la forme des entrées et sorties de Ac2lus : ce sont des courbes représentant le nombre d'événements

(formalisme du Real-Time Calculus [5]) entrant (ou sortant) dans le programme lustre en fonction de l'intervalle de temps, il y a en réalité deux courbes (FIG. 2) : une représentant le minimum d'événements possible se produisant sur un intervalle de temps et une représentant le maximum. Ces courbes sont représentées en temps discret.
Comme représenté sur la Fig. 1, ac2lus a pour entrée et sortie ce type de courbe, on utilise un *adapter* (qui est en réalité un observateur) afin de transformer les contraintes exprimées par les courbes en propriétés exploitables et pour s'assurer que les entrées du composant LUSTRE respectent bien ces contraintes. Le *back adapter* est également un observateur, il génère des courbes de sortie en observant le composant lustre et ses sorties en fonction de ses entrées

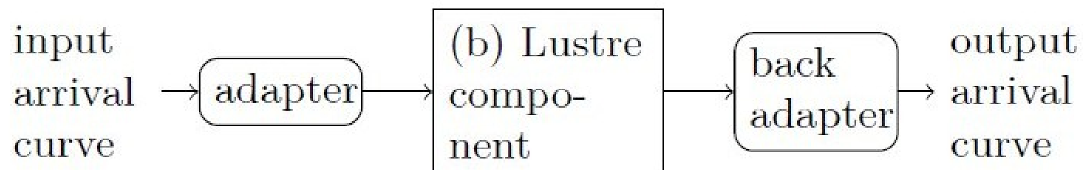


FIGURE 1 – Ac2lus [5]

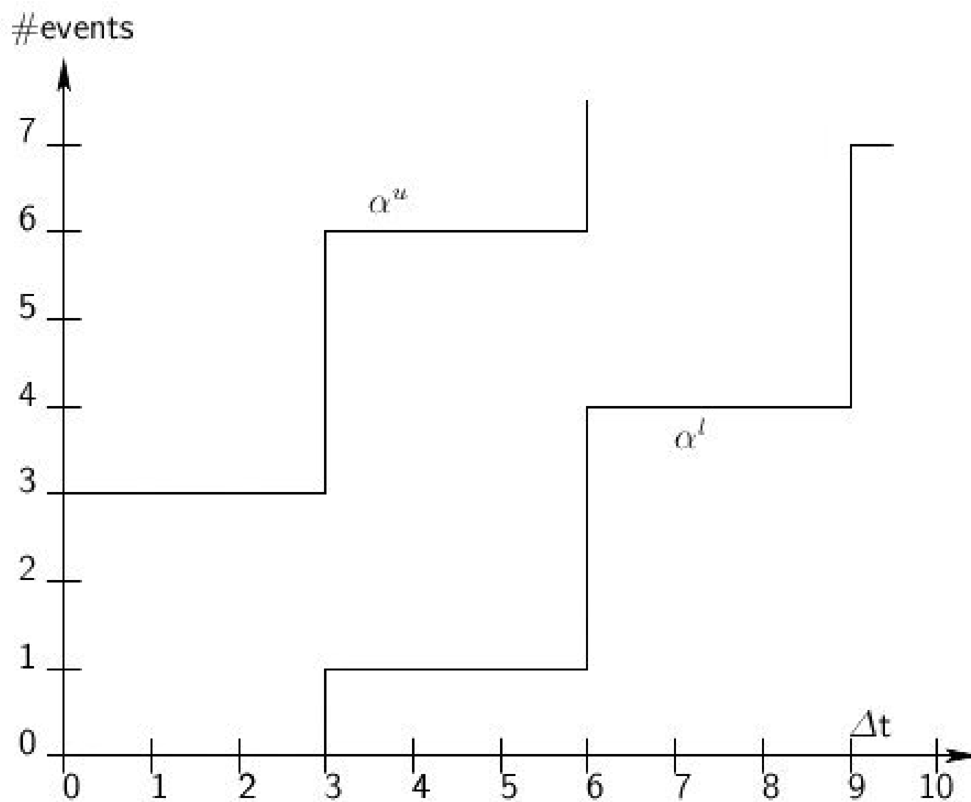


FIGURE 2 – courbes d'entrées/sorties [5]

Commentaire sur les courbes : On représente le nombre d'événements en fonction du temps, la courbe $\alpha^l(\delta)$ représente le minimum d'événements, et la courbe $\alpha^u(\delta)$ le maximum sur un intervalle de temps. Sur l'exemple (Fig. 2) on voit que :

– sur [t,t+3[$\forall t$, le nombre minimum d'événements est 0 et le maximum est 3, sur tout intervalle de taille 3, on aura ces même extrémités.
– sur [t,t+6[$\forall t$, le nombre minimum est 1 et le maximum est 6.

2.2.2 Obtention des courbes d'arrivées

Comment obtenir les courbes d'arrivées ?

Revenons d'abord sur le *back adapter*, c'est lui qui nous permet de générer les courbes de sorties, comment fonctionne t'il ?
C'est en fait un observateur prenant en compte les entrées du programme, le programme et ses sorties. L'objectif est de trouver, pour chaque taille d'intervalle, le nombre minimum et maximum d'événements sortant du programme LUSTRE, les points ne sont pas trouvés directement, une recherche dichotomique est nécessaire, en effet, pour chaque point de la courbe, on va essayer plusieurs valeurs et tester si elles correspondent ou non aux événements sorties du programme. On ne construit donc pas directement la courbe à partir des résultats obtenus mais on construit d'abord une courbe, puis on la compare aux résultats, et si elle ne correspond pas, on en essaye une autre.
On va pour cela utiliser des outils de preuves (présentés dans la partie suivante), en les interrogeant sur le point courbe de sortie à tester en les interrogeant sur des propriétés, par exemple : *est ce que 2 est un majorant du nombre d'événements sortants sur l'intervalle ?*, les outils répondrons Oui, Non ou Ne Sait Pas.
Pour trouver la valeur recherchée, on procède par dichotomie : par exemple, si on cherche le maximum sur un intervalle de taille 1 :

1. On recherche tout d'abord une valeur X_n pour laquelle les outils ont répondu *Oui*, il ne peut donc pas avoir plus de X_n événements en sortie. Dans cette phase de recherche, lorsque les outils répondent *Non* ou *Ne Sait Pas* pour un test avec une valeur Y, la valeur suivante à être testée est 2*Y+1. La première valeur testée étant 0.
2. Une fois la première étape terminée et la valeur X_n trouvée, nous recherchons maintenant la plus petite valeur X telle que les outils répondent *Oui* au test *X est un majorant du nombre d'évènements sur l'intervalle* en effectuant une recherche dichotomique.

On trouve ainsi les courbes maximum et minimum. Cette méthode appliqué jusqu'à présent n'utilise que les outils Kind [3] et nbac [4] en exécutant d'abord Kind et si il n'apporte pas de réponse au bout d'un certains délai, Nbac est exécuté.
La Fig. 3 représente la manière dont les outils de preuve sont utilisés afin d'obtenir les courbes de sorties. La zone en pointillé de la figure représente la partie en projet d'ac2lus, nous nous intéressons à la branche qui utilise aspic en tant que outil de preuve. On notera que les tests faits avec les outils de preuve sont effectués sur un fichier généré avec un point à tester, il y a en réalité un nouveau fichier généré pour chaque point mais ils sont très semblables, seule la valeur du point testé change. A la fin du test, soit la recherche est terminée et la courbe de sortie est renvoyée, soit on recommence un test avec un nouveau point, d'où les boucles sur la Fig. 3 dans les cas d'utilisation d'outils de preuve. Nous verrons dans la partie suivante que Aspic renvoie également des invariants sur le programme testé, ce qui nous permettrais de ne plus faire de boucle, et combiné à une analyse performante de ceux-ci, pourrait améliorer les performances de Ac2lus.

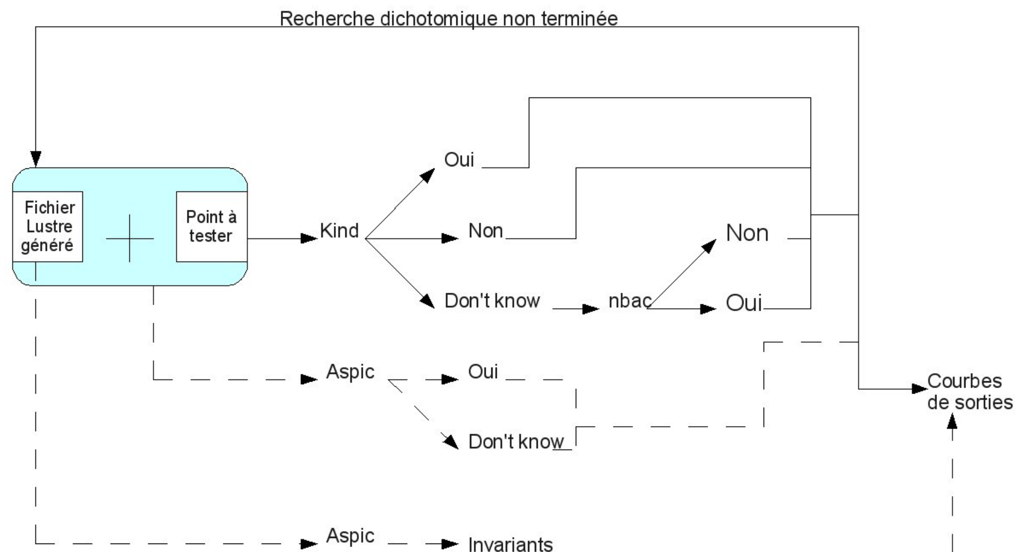


FIGURE 3 – Ac2lus - obtention des courbes d'arrivées

2.3 Outils

Nous présentons ci dessous les outils de preuve permettant de trouver les points des courbes de sorties lors de l'exécution de ac2lus. Pour le moment, Ac2lus fonctionne (par défaut) en exécutant d'abord kind avec un délai de 10s et si aucune réponse n'est obtenue au bout de ce délai, nbac est exécuté. Rappelons que l'objectif est de parvenir à utiliser aspic qui , grâce a des méthodes d'accélérations, à théoriquement de meilleures performances.

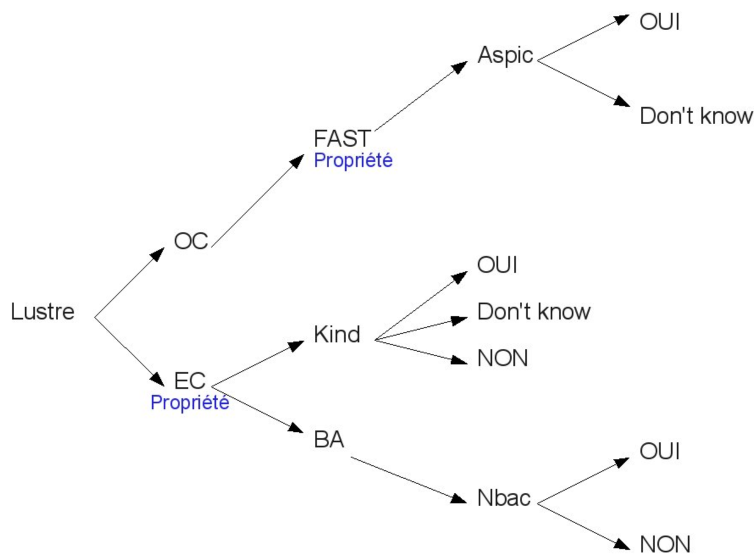


FIGURE 4 – outils de preuve

2.3.1 Kind

Kind [3] est un outil de preuve répondant Oui, ne sait pas ou Non à une propriété donnée pour un programme donné. En réalité, soit kind parvient à prouver la propriété la réponse est oui, soit la propriété est fausse et kind renvoie des contre-exemples, soit kind par en boucle infinie car il ne parvient pas prouver la propriété ni a trouver de contre exemple, la réponse est donc Ne Sait Pas.
Kind présente l'avantage de pouvoir déclarer la propriété à tester directement comme suit :

```
node Noeud(in:int) returns(out:int);
let
--PROPERTY B;
  .
  .
  .
tel
```

Kind tentera de prouver que l'expression booléenne (ou le booléen) B placé après $--PROPERTY$ est toujours vrai ou donnera un contre exemple représenté par la valeur de chacune des variables du programme lors d'une exécution ou B est faux.
On note que la propriété doit être placé dans le fichier .ec.

2.3.2 Nbac

Nbac [4] est un outil de preuve répondant Oui ou Non (à interpréter comme Ne Sait Pas) et s'appuyant sur des automates via le format intermédiaire .ba. Nbac tente de répondre à la question :*le premier booléen en sortie est il toujours vrai ?*, c'est à dire que Nbac vérifiera que ce booléen est toujours vrai, si il peut devenir faux lors d'une exécution du programme, alors il répondra Non. Voici un exemple d'utilisation :

```
node Noeud(in:int) returns(Ok_Nbac:bool);
let
Ok_nbac= true -> pre(B);
  .
  .
  .
tel
```

On pourra faire la traduction d'un fichier Lustre à un fichier .ba à l'aide de l'outil *lus2nbac* (*lus2nbac Mon_fichier MainNode*) puis exécuter nbac sur le fichier généré (*nbacg.opt MainNode.ba*). Pour que nbac fonctionne correctement, nous devons faire appel à l'opérateur *pre* dans le fichier Lustre.

2.3.3 Aspic

Pour plus d'informations, lire la thèse de Laure GONNORD [2]
Aspic est un outil de preuve s'appuyant sur des automates, il donne aussi des invariants pour chaque états. Les automates sont décrits dans des fichiers au format FAST. Un fichier FAST est constitué de différentes parties :

– Déclaration des variables et de tous les états de l'automate
– Description de chaque transition.
– Déclaration de l'état initial ainsi que des 'bad regions' qui serviront dans un but de preuve (on prouvera que ces régions ne sont jamais atteintes par exemple) la déclaration de bad région est facultative

Aspic nous donnera donc des invariants pour chaque état de l'automate décrit dans le fichier FAST ainsi qu'une réponse (OUI/Ne sait pas) sur la propriété à prouver si une bad region est déclarée.

On décrit brièvement ci dessous les méthodes d'accélération de Aspic :
Les accélérations vont permettre d'améliorer la précision ainsi que les performances de Aspic, elles se font principalement au niveau des boucles, en formant, à partir du point de contrôle ou se situe la boucle, deux points de contrôles dont la transition est un invariant d'itération de la boucle initiale.
On joint une illustration de cette méthode (FIG. 5) :

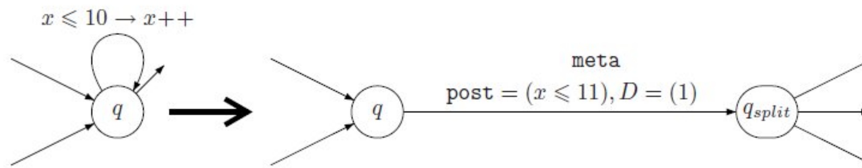


FIGURE 5 – accélération aspic [2]

3 Vers l'utilisation de aspic dans AC2lus

3.1 Expérimentation aspic

3.1.1 De LUSTRE vers ASPIC

Nous allons illustrer au travers d'exemples comment utiliser aspic à partir d'un fichier LUSTRE. Rappelons que Aspic prend en entrée un fichier au format .fst (FAST) et qu'il n'existe pas à l'heure d'aujourd'hui de programme effectuant la transition LUSTRE vers FAST. Cependant, il existe divers programmes qui vont nous permettre d'effectuer le lien entre LUSTRE et ASPIC. L'outil oc2fst [2] permet de faire le lien entre le format oc et fst alors qu'un fichier oc peut être obtenu à partir d'un fichier LUSTRE lors de sa compilation ou grâce à l'outil lus2oc.

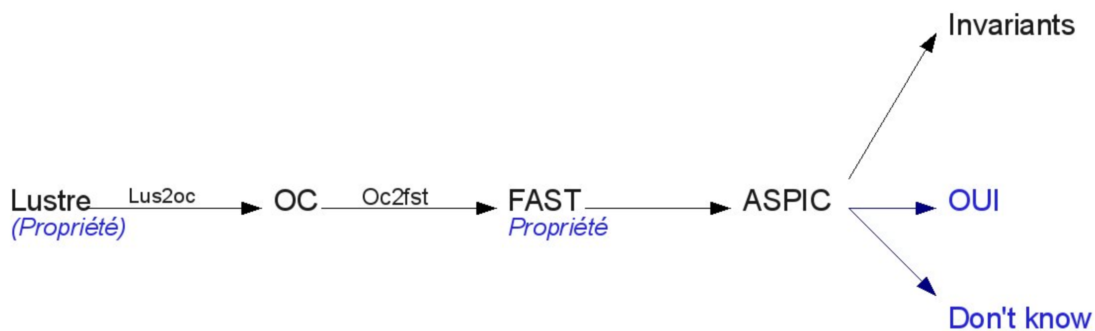


FIGURE 6 – Chaîne Lustre vers Aspic

3.1.2 Exemple

Nous présentons ci dessous un exemple de la chaîne LUSTRE vers aspic ainsi que la comparaison des résultats obtenus avec kind,nbac et aspic.

```
node MAX(a,b:int) returns (c:int);

let
        c=if a<b then b else a;
tel;
```

FIGURE 7 – node exemple

On propose d'étudier l'algorithme FIG. 7, qui a pour entrées deux entiers *a* et *b* et qui renvoie leur maximum *c*.

Test de propriétés

On propose à présent d'effectuer le test de la propriété a+b<=2c avec les outils kind, nbac et aspic.

-> **kind et nbac**

Pour kind et nbac, nous devons modifier le noeud (Fig. 8) pour nous permettre de tester cette propriété. On joint le fichier .ec issu de ce noeud.

```
node MAX_test(a: int; b: int) returns (Ok_nbac: bool;Ok: bool);
var c: int;
let
--%PROPERTY Ok;
  Ok_nbac = (true -> (pre Ok));
  Ok = (a+b <= 2*c);
  c = (if (a < b) then b else a);
tel
```

FIGURE 8 – node de test pour kind et nbac

– Pour **kind**, on ajoute —% *PROPERTY Ok ;* , ce qui signifie kind va chercher des contre exemples ou le booléen *OK* deviendrait faux à au moins un instant de l'exécution (donc ou a+b>2c), voici le résultat obtenu (*kind MAX.ec*) :

```
1 step is Valid.
+++ The property holds. +++
Done.
```

On constate donc que kind n'a pas trouvé de contre exemple, la propriété est correcte.

– rappelons que **nbac** tente de prouver que le premier booléen en sortie de programme est toujours vrai, d'où l'ajout du booléen *Ok_nbac* qui, pour des raisons de syntaxe liées au format .ba, vaut vrai à la première exécution puis la valeur précédente de *Ok*, donc si *Ok* devient faux, *Ok_nbac* deviendra faux à l'exécution suivante.
On exécute alors les commandes suivantes : *lus2nbac MAX_test.ec MAX_test* qui génère un fichier MAX_test.ba, puis *nbacg.opt MAX_test.ba*. On obtient le résultat suivant :

```
SUCCESS: property proved
```

-> **Aspic**

décomposons maintenant la chaîne qui va nous permettre d'utiliser Aspic :

1. On exécute *lus2oc exemple.lus MAX* qui va générer un fichier MAX.oc qui contient l'automate représentant le noeud MAX.
2. *oc2fst MAX.oc*[2] qui va générer un fichier MAX.fst contenant également un automate mais représenté par une structure différente de celle du .oc avec la possibilité de définir une 'bad region'.
3. Nous n'avons plus qu'à exécuter *aspic MAX.fst*.

Afin de tester la propriété, nous devons éditer la *strategy* du fichier MAX.fst (Fig. 9 et entrer en 'region bad' : a+b>2c (!(a+b<=2c) génère une erreur syntaxique), le programme vérifiera alors si il est possible d'atteindre ou non cette région, si la region est inatteignable, alors a+b<=2c. Il faut également compléter

```
Region init := {state=loc_0 && a<=c && b<=c};

Region bad := {a+b>2c};
```

FIGURE 9 – strategy - MAX.fst

l'état initial et ajouter des conditions sur c. On peut alors exécuter *aspic MAX.fst* : le résultat est donné en Annexe A. On observe que les invariants rendus sont cohérents avec le programme mais aussi le résultat du test :

```
****** Result : All the bad locations are unreachable ******
```

Ce qui signifie donc que a+b>2c est inatteignable, et que par conséquent, a+b<=2c. Toutefois, on notera également lors de cette expérience un certain nombre de bug (surtout des problèmes de Oc2fst, typiquement la génération de code générant des erreurs syntaxiques dans aspic) qui ont été contourné lors de cet exemple mais seront détaillés dans la partie 3.3 et constitueront un frein à l'utilisation de aspic dans ac2lus mais également le fait qu'il n'y à pas de moyen de d'exprimer une propriété dans le fichier Lustre et de la conserver tout au long de la chaine.

Évolution de l'automate au cours de la chaîne On se propose d'étudier la structure d'un automate représentant un programme via un exemple :
Nous changeons de noeud pour cet exemple car le noeud MAX ne génère pas d'automate intéressant, en effet, on conçoit facilement qu'il ne contiendrait qu'un unique état. Pour cela, nous utiliserons donc le noeud suivant :

```
node VersCent(ini: int) returns (B: bool;C: int);
let
  B = (true -> ((pre C) = 100));
  C = ini ->    if (pre C > 100) then (pre C) - 1
                else    if (pre C < 100) then (pre C) + 1
                        else (pre C);
tel
```

Ce noeud a pour entrée un entier *ini* et pour sortie un Booléen *B* et un entier *C*, B vaut *true* ssi la valeur précédente de *C* vaut 100, *C* est initialisé à *ini*, puis, si la valeur précédente de C est strictement supérieur à 100, on lui soustrait 1, si elle est strictement inférieur à 100, on lui ajoute 1, sinon (c'est à dire quand C=100) on ne modifie pas C.

1. on traduit ce noeud en fichier .oc grâce à *lus2oc*. On peut à cette étape visualiser l'automate obtenu en générant un fichier .atg avec l'outil *oc2atg* puis et en l'affichant grâce à la commande *atg*.

   On observe un automate relativement simple, contenant deux états dont une boucle.
2. On génère ensuite un fichier Fast à l'aide de *oc2fst* puis on exécute *aspic* avec l'option *-printdot* qui nous génère un fichier .dot contenant l'automate issu de cette exécution. L'automate est joint en annexe B (5.2).

   On observe ce coup ci un automate complexe,pourquoi :
   - *oc2fst* génère un automate non déterministe ce qui est causé notamment par l'absence de variable booléenne dans le format Fast, en effet, les booléens sont transformés en points de contrôles, ce qui contribue également à en augmenter le nombre et rend l'automate plus complex.
   - Les méthodes d'accélération de *aspic* illustrées dans la partie 2.3.3 contribuent à augmenter le nombre d'états de l'automate et a le rendre plus complexe.

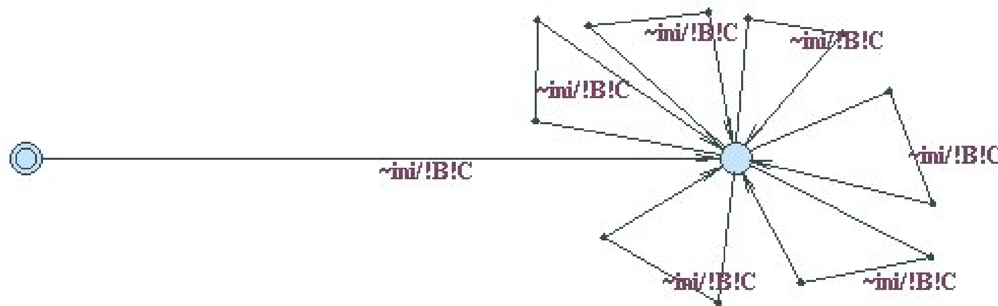


FIGURE 10 – automate après lus2oc

Nous pensons toutefois que des améliorations peuvent être apportées au niveau de oc2fst.

3.2 Expérimentation - AC2LUS

L'objectif est de tenter d'utiliser, de manière non automatique, aspic dans AC2LUS (à la place de kind et nbac). Lors de l'obtention des courbes de sorties et la recherche dichotomique, pour chaque point testé, un fichier (binary_search.ec) est généré puis, les outils de preuve sont exécutés sur ce fichier. Ce fichier est généré de façon à ce que kind et nbac fonctionnent correctement. On joint en annexe c (5.3) un fichier auto-généré.
Le fichier généré contient deux sorties, toutes deux booléennes, et notons que :

- oc2fst transforme les variables booléennes en points de contrôle de l'automate généré, aucune variable booléenne ne figure donc dans le fichier fast après la transformation oc2fst.
- il n'y a pas de possibilité d'exprimer une propriété dans un noeud lustre et de la retrouver dans le fichier généré fast.

On ne peut donc pas effectuer de test sur le fichier binary_search initialement généré car, il est très difficile de définir la bonne propriété dans un tel fichier FAST.
Une idée fut de tenter de faire générer un état puits dans l'automate final obtenu (celui contenue dans le fichier fast), nous y sommes arrivés avec le noeud ci dessous :

```
node MAX_mm(a: int;b: int) returns (out: int);
var c: int;
OK : bool;
let
  out = if OK then 1 else 0;
  OK = (a <= c) and (true -> pre(OK));
  c = MAX(a,b);
tel;
```

En effet, on met en sortie un entier qui sera donc conservé lors de la chaîne Lustre vers Fast, de plus, il vaut 1 lorsque OK est vrai et sinon, ce qui nous permettra de repérer les états et transitions vers lesquels la propriété testée est fausse. On note également que le fait que le booléen soit tout le temps faux du moment qu'il a été faux une fois nous permet de générer cet état puits. Pour tester la propriété avec aspic, nous n'avons plus qu'a compléter la Bad region en y indiquant cet état.
Malheureusement, ce genre de mise en place a échoué avec le fichier Binary_search généré par Ac2lus car Oc2fst ne génère pas systématiquement des automates déterministes, nous ne sommes donc pas arrivés à faire généré d'état puits ;

3.3 Problèmes rencontrés

On liste ci dessous les problèmes et difficultés rencontrés lors de la réalisation de la chaîne Lustre vers aspic dans l'optique d'une intégration dans Ac2lus. La majorité est due à l'outil oc2fst.

- Il n'existe pas de moyen de conserver une propriété à tester tout au long de la chaîne : il est impossible de définir une propriété dans un fichier lustre, puis après les traductions vers le fichier fast, de retrouver cette propriété en *bad region* dans le FAST (la règle *le premier booléen en sortie est toujours vrai* n'est pas implémenté). De plus les conditions initiales ne survivent pas non plus à la chaîne.
- Des erreurs de compatibilité existent entre oc2fst et aspic : celle que nous avons le plus fréquemment rencontré est la suivante :

```
action  := output (a<=a);
```

FIGURE 11 – code généré par oc2fst

Aspic générant une erreur syntaxique en analysant ce code.

4 Conclusion

4.1 Résumé

L'objectif était d'étudier l'utilisation de Aspic dans Ac2lus, il fallait donc pour cela comprendre le fonctionnement et l'utilité de Ac2lus, cela passe d'abord par un apprentissage et une initiation au langage Lustre 2.1, rappelons que c'est un langage synchrone développé par le laboratoire Verimag initialement conçu pour l'implémentation mais également utilisé aujourd'hui pour la simulation. Il faut ensuite s'intéresser au fonctionnement de Ac2lus, au sens large avec le fonctionnement global, au plus précis avec la compréhension des courbes en entrées et sorties du programme mais également comment les obtenir : avec la recherche dichotomique et l'utilisation d'outils de preuve de propriété (Kind et Nbac). On comprend alors ou va se situer Aspic et comment il pourrait nous être utile, on peut donc expérimenter son fonctionnement, notamment en étudiant comment il peut être utilisé à partir d'un fichier Lustre puis dans Ac2lus.

Bien que cela soit théoriquement possible, nous n'avons pas obtenu de résultat satisfaisant avec l'utilisation de Aspic dans Ac2lus, il y a plusieurs raisons à cela, Aspic est plus axé sur le renvoi d'invariants décrivant un programme que sur la preuve de propriété, nous détournons donc plusieurs outils de leurs principale fonction, dont Oc2fst (outil présent dans la chaîne permettant d'exécuter aspic à partir d'un fichier Lustre) qui ne nous permet pas à l'heure actuelle d'exprimer correctement et facilement des propriétés à tester, de plus il ne semble pas maintenu par rapport à Aspic et génère parfois du code non interprétable par ce dernier. On notera toutefois que nous avons obtenu des résultats satisfaisants sur des exemples faciles de l'utilisation de Aspic dans Ac2lus en trouvant des moyens de contourner ces problèmes, le but finale étant une automatisation de Ac2lus avec Aspic, nous n'avons pas, pour le moment, donné suite à cette expérimentation car cette opération n'est pas possible avec les outils existants.

4.2 Perspectives

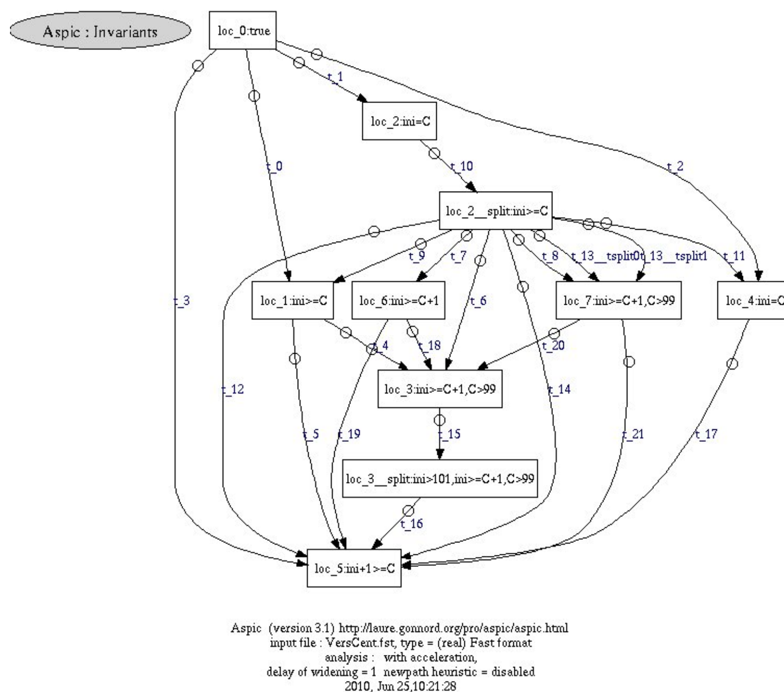
La majorité des problèmes rencontrés provient de la chaîne Lustre vers Fast (le format d'entrée de Aspic), aspic fonctionne correctement avec un bon fichier Fast mais la chaine génère des erreurs et ne nous permet d'exprimer correctement les propriétés souhaitais, une nouvelle version de oc2fst devrait bientôt voir le jour ce qui devrait nous permettre d'automatiser l'utilisation de Aspic dans Ac2lus à condition que la propriété ne soit pas déjà perdu lors de la transition *lus2oc* .
L'avantage majeur de aspic est que, en plus de prouver des propriétés, aspic renvoie des invariants sur le programme, à terme, l'idée serait de récupérer ces invariants et de les utiliser pour obtenir directement une propriété sur la sortie du programme, ce qui devrait améliorer les performances de Aspic, car la recherche dichotomique ne serait plus nécessaire dans la plupart des cas.

5 Annexes

5.1 Annexe A : Résultat de l'exemple Aspic

```
***
Aspic by Laure Gonnord, version 3.1
Binary compiled on Monday 15 March 2010
Input file = MAX.fst
***
   * Name of model = MAX, Name of file = MAX.fst
   * 3 variable(s) and 3 locs(s) and 8 transitions(s) in the .fst
         ## we have no bad node, just a formula
   * The initial state has label loc_0 and the associated polyhedron is {
$>=0,1>0,a<=c,b<=c}
   * There is/are 1 bad node(s)
   * Parsing OK, now analysing
   * LRA with acceleration with delay =  1

   [...]

join = {b<=c,1>0,$>=0,a<=c}
join initial with 2 new polys
resu = {a<=c,b<=c,$>=0,1>0}
  ****** Result : All the bad locations are unreachable ******

 * Invariants =

loc_0 -----> {b<=c, a<=c}

loc_1 -----> {false}

loc_2 -----> {b<=c, a<=c}

__bad_0 -----> {false}

loc_1__split + loc_1 ==
 {false} U
   {false}
    = {false}

loc_2__split + loc_2 ==
 {b<=c, a<=c} U
   {b<=c, a<=c}
    = {b<=c, a<=c}

 * Stats =  { time=0.000000; iterations=3; descendings=0; }

 Finished !
```

5.2 Annexe B : Automate issue de aspic du noeud VersCent



5.3 Annexe C : Fichier binary_search généré par ac2lus

```
node binary_search
  (in_seq_Ainput: int)
returns
  (OK_for_nbac: bool;
  OK: bool);

var
  V107_obs_ok_Ainput: bool;
  V109_obs_ok_Zoutput: bool;
  V179_ok_now_points: bool;
  V180_inter1: bool;
  V193_ok_now_points: bool;
  V194_inter1: bool;

let
--%PROPERTY OK;
  OK_for_nbac = (true -> (pre OK));
  OK = ((not V107_obs_ok_Ainput) or V109_obs_ok_Zoutput);
  V107_obs_ok_Ainput = (V179_ok_now_points and (true -> (pre V107_obs_ok_Ainput
  )));
  V109_obs_ok_Zoutput = (V193_ok_now_points and (true -> (pre
  V109_obs_ok_Zoutput)));
  V179_ok_now_points = ((0 <= in_seq_Ainput) and (in_seq_Ainput <= 10));
  V180_inter1 = (true -> false);
  V193_ok_now_points = (in_seq_Ainput <= 3);
  V194_inter1 = (true -> false);
tel
```

Références

[1] http ://www-verimag.imag.fr. 1.1

[2] Laure Gonnord. *Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires*. Thèse de doctorat, Université Joseph Fourier, Grenoble, October 2007. 1.2, 2.3.3, 5, 3.1.1, 2

[3] George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *FMCAD*, 2008. 2.2.2, 2.3.1

[4] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 2003. 2.2.2, 2.3.2

[5] Matthieu Moy and Karine Altisen. ac2lus : Bringing smt-solving and abstract interpretation techniques to real-time calculus through the synchronous language lustre. Technical Report TR-2010-2, Verimag Research Report, 2009. 2.2, 2.2.1, 1, 2

[6] Nicolas HALBWACHS Pascal RAYMOND. *A TUTORIAL OF LUSTRE*, August 2007. 2.1