Un module prolog de mu-calcul booléen : une réalisation par BDD

Serge Colin*, Fred Mesnard* et Antoine Rauzy**[1]

Résumé : De nombreux problèmes tels que l'analyse statique de programmes et la vérification symbolique nécessitent des calculs de points fixes. Des outils spécialisés existent, mais manquent de souplesse. Les langages de programmation logique avec contraintes offrent une bonne expressivité mais ne permettent ni la quantification sur les prédicats, ni la quantification universelle sur les variables. L'idéal serait donc de bénificier des deux. Ce papier présente un module Prolog intégrant le mu-calcul sur les booléens et basé sur les diagrammes binaires de décision.

Mots-clés : Plc, Prolog, mu-calcul booléen, calcul de point fixe

1. Introduction

De nombreux problèmes tels que l'analyse statique de programmes et la vérification symbolique de circuits nécessitent des calculs de points fixes et donc une forme de quantification d'ordre supérieur sur des relations. Des outils spécialisés tels que SMV [MCM 92] pour la vérification symbolique ou des interpréteurs spécifiques tels que Toupie [COR 97] existent, mais du fait de leur spécialisation, manquent de souplesse. D'un autre côté, les langages de programmation logique avec contraintes (CHIP [DIN 88], Prolog III [COL 90], CLP(R) [JAF 87]) offrent une bonne expressivité, mais ne permettent ni quantification sur les prédicats, ni quantification universelle explicite sur les variables. De plus, ces langages permettent généralement de trouver une solution

[1]*IREMIA, Université de la Réunion, 15, avenue René Cassin - BP 7151 - 97715 Saint Denis Messag. Cedex 9 France, email : {colin,fred}@univ-reunion.fr, web : www.univ-reunion.fr/~gcc. ** LaBRI, URA CNRS 1304 Univeristé Bordeaux I, 351, cours de la Libération 33405 Talence, Cedex, France, email : rauzy@labri.u-bordeaux.fr

à un problème (sous forme de contraintes), éventuellement la meilleure selon un certain critère, alors que l'on préférerait parfois exprimer l'ensemble des solutions en une unique contrainte.

Notre travail a consisté à tenter d'allier l'efficacité de ces outils spécialisés à la souplesse des environnements de programmation logique avec contraintes. Plus précisément, de réaliser un module SICStus Prolog gérant un nouveau type de contraintes «à la Toupie» portant sur les relations booléennes. Ce module implante le μ-calcul sur les booléens et pour une bonne efficacité utilise les diagrammes binaires de décision. Son code réutilise les primitives de gestion des diagrammes binaires de décision définies dans le module `clpb` des contraintes booléennes de SICStus [CAR 94].

Ce papier s'organise ainsi : dans une première section, nous introduirons le formalisme classique du μ-calcul sur les booléens, illustré par des exemples. Nous présenterons ensuite la structure de données diagramme binaire de décision. Après avoir décrit notre implantation, nous reprendrons les exemples décrits précédemment en montrant de quelle façon notre module permet de les traiter. Nous fournirons notamment des temps de calcul pour l'analyse de clôture. Enfin, dans une dernière partie, nous discuterons des développements possibles de ce travail.

2. Le μ-calcul

Le μ-calcul propositionnel est une logique qui a été introduite par Park [PAR 74], basée sur le calcul de points fixes, permettant d'exprimer de nombreuses propriétés des systèmes de transition (par ex. «étreinte mortelle» (deadlock), états atteignables) et strictement plus expressive que des logiques temporelles comme CTL (voir [MCM 92, BUR 92] pour plus de détails).

2.1. *Définitions*

En plus des variables, constantes, connecteurs et quantificateurs du calcul propositionnel, le μ-calcul permet de définir des relations comme plus grands ou plus petits points fixes d'équations, fournissant donc une forme de quantification d'ordre supérieur sur les relations.

Voici une définition formelle, tirée de la thèse de McMillan [MCM 92]. On distingue deux types de formules : booléennes et relationnelles, ainsi que deux types de variables : individuelles (par exemple x : un état) et relationnelles (par exemple R : la relation de transition). Un modèle pour le μ-calcul est un

triplet $M = (S, \phi, \psi)$ où S est un ensemble fini d'états, ϕ est une *interprétation individuelle* (application de chaque variable individuelle vers un état de S), et ψ une *interprétation relationnelle* (application de chaque variable relationnelle n-aire vers un sous-ensemble de S^n). On définit les formules booléennes de manière usuelle par induction :

1. *vrai* et *faux* sont des formules booléennes.
2. Si p et q sont des formules booléennes, alors $p \vee q$, $\neg p$ le sont aussi (les autres opérateurs booléens peuvent être définis à partir de $\neg$ et $\vee$).
3. Si p est une formule booléenne et x une variable individuelle, $\exists x.p$ et $\forall x.p$ sont des formules booléennes.
4. Si R est une formule relationnelle n-aire et $(x_1, \ldots, x_n)$ un vecteur de variables individuelles, alors $R(x_1, \ldots, x_n)$ est une formule booléenne.

Les formules relationnelles sont définies comme suit :

1. Toute variable relationnelle n-aire R est une formule relationnelle.
2. Si p est une formule booléenne et $(x_1, \ldots, x_n)$ un vecteur de variables individuelles , $\lambda(x_1, \ldots, x_n).p$ est une formule relationnelle.
3. Enfin, si R est une variable relationnelle, et F une formule relationnelle monotone en R, alors, $\mu R.F$ (plus petit point fixe) et $\nu R.F$ (plus grand point fixe) sont des formules relationnelles.

Une formule relationnelle F est dite monotone en R si la variable relationnelle R apparaît seulement dans un nombre pair de négation à l'intérieur de F.

La formule relationnelle (cf. exemple 2 section 2.2)

$$\mu R.\lambda(x).[G(2, x) \vee \exists y.(R(y) \wedge G(y, x))]$$

est donc un exemple de formule valide (avec plus petit point fixe), sachant que R et G sont des variables relationnelles et x et y des variables individuelles.

Le théorème de Tarski [TAR 55] nous donne un algorithme pour calculer un point fixe ainsi que les conditions de validité de ce calcul (monotonie) :

- pour calculer le plus petit point fixe d'une fonction monotone f, on part de l'ensemble (relation) vide $\emptyset$ et on itère ($f(\emptyset)$, $f^2(\emptyset) = f(f(\emptyset))$, ...), jusqu' à ce que l'on arrive au point fixe (*i.e.* $f^{n+1}(\emptyset) = f^n(\emptyset)$).
- pour calculer le plus grand point fixe, on procède de manière symétrique. Le plus petit et le plus grand point fixe, en effet, sont duaux : on a $\nu X.f(X) = \neg\mu X.\neg f(X)$. On partira donc de l'ensemble tout entier, en retirant des éléments à chaque itération au lieu d'en ajouter.

Enfin, dans un modèle donné (S, ϕ, ψ), on identifie la variable relationnelle R avec la relation $\psi(R)$. La formule $\lambda(x_1, \ldots, x_2).p$ correspond à l'ensemble

des n-uplets $(x_1, \ldots, x_2)$ tels que p soit vraie. La formule $\mu R.F$ correspond au plus petit point fixe de l'opérateur $\tau = \lambda R.F$, tandis que $\nu R.F$ correspond à son plus grand point fixe.

2.2. *Exemples d'utilisation du μ-calcul*

Considérons l'automate A défini figure 1 et appelons G sa relation de transition.

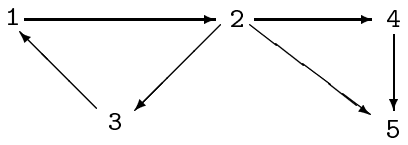

Figure 1. *L'automate A*

Nous allons illustrer par quelques exemples la caractérisation de propriétés de cet automate avec le μ-calcul :

exemple 1 *On peut chercher à calculer la fermeture transitive REACH de cet automate, REACH étant une relation binaire :*

$$REACH = \mu Y.\lambda(x,y).[G(x,y) \vee \exists z.\,(Y(x,z) \wedge G(z,y))]$$

En effet, un couple d'états (x,y) est dans la fermeture transitive si il existe une transition de x à y ou s'il existe un état z tel que les couples (x,z) et (z,y) appartiennent à la fermeture transitive.

exemple 2 *On peut calculer la relation unaire ACCDEP2 des états accessibles depuis l'état 2 :*

$$ACCDEP2 = \mu R.\lambda(x).[G(2,x) \vee \exists y.(R(y) \wedge G(y,x))]$$

Autrement dit, un état x est accessible depuis l'état 2 s'il existe une transition de l'état 2 à l'état x ou s'il existe une transition d'un état accessible depuis l'état 2 vers l'état x.

exemple 3 *On peut chercher les états conduisant à un* «dead-lock» *(étreinte mortelle) :*

$$DL = \mu Z.\lambda(x).[\forall t.G(x,t) \Rightarrow Z(t)]$$

Ce sont tous les états qui mènent obligatoirement à un état à partir duquel aucune transition n'est plus possible. La relation sera initialisée par tous les états n'ayant pas de successeurs (l'implication $A \Rightarrow 0$ devant être vraie, A doit être faux donc $G(x,t)$ doit être faux).

exemple 4 *On peut chercher les états qui ne sont pas prédécesseurs de l'état 4 :*

$$NP4 = \nu Z.\lambda(x).[\forall t.(G(x,t) \Rightarrow ((t \neq 4) \wedge Z(t))]$$

i.e.*un sommet n'est pas un prédécesseur du sommet 4 s'il n'a pas de successeur ou si toute transition qui en part mène à un sommet n'étant pas prédécesseur de 4. On a ici un plus grand point fixe car l'idée est que l'on va éliminer (filtrer) successivement tous les états qui menaient à l'état 4 (la trace est :* $\{1,2,3,4,5\}$, $\{1,-,3,4,5\}$, $\{-,-,3,4,5\}$, $\{-,-,-,4,5\}$, *où '-' figure un élément retiré à la ième itération). On aurait bien sûr pu aussi utiliser un plus petit point fixe (par dualité).*

Remarquons que comme dans Toupie [COR 97], ce μ-calcul est étendu en autorisant le nommage des relations et donc les systèmes d'équations de points fixes.

3. Les diagrammes binaires de décision

Les diagrammes binaires de décision (BDDs) [BRY 86, BRY 92], sont une structure de donnée compacte et efficace pour représenter symboliquement les fonctions booléennes. Ce sont des graphes acycliques orientés représentant une information factorisée.

L'idée de cette représentation est que si l'on regarde l'arbre de décision binaire d'une formule booléenne (les noeuds sont les variables, la branche gauche correspond à une valeur de 0 pour la variable et la branche droite à une valeur de 1, cf. figure 2), on s'aperçoit que certains sous-arbres sont identiques (notamment les feuilles 0 et 1), et que que l'on peut donc réduire certaines parties de l'arbre, obtenant alors un graphe acyclique «compact». Il est aussi possible d'éliminer certains noeuds s'ils n'influent pas sur la valeur de vérité de la formule. Ainsi, on peut supprimer le noeud x_3 le plus à gauche dans l'arbre de la figure 2 : si $x_1 = 0$ et $x_2 = 0$, quelque soit la valeur de x_3, la formule sera fausse.

Enfin, on peut améliorer cette représentation en se donnant un ordre sur variables : on parle alors de diagrammes binaires de décision ordonnés (OBDDs). Par la suite, nous utiliserons indifféremment le terme de BDDs pour les BDDs et les OBDDs. Pour un ordre donné sur les variables, la représentation

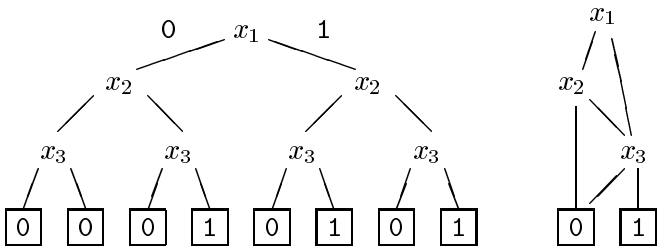

Figure 2. *L'arbre de décision T représentant la formule $(x_1 \vee x_2) \wedge x_3$ et l'OBDD associé pour l'ordre $x_1 < x_2 < x_3$*

est canonique. Cet ordre sur les variables a bien sûr une grande influence sur la forme et la taille des BDDs que l'on obtient : un «mauvais» ordre peut rendre la taille de la structure exponentielle. De plus, en étiquetant certains arcs du graphe comme négatifs (dans un parcours, chaque fois que l'on passe par un arc négatif, on inversera la valeur des feuilles), mais en imposant toutefois certaines restrictions pour assurer la canonicité de la représentation, on peut encore améliorer la compacité de la représentation. On notera que bien que les complexités théoriques des algorithmes opérants sur les BDDs ne soient pas très bonnes, dans la pratique, ceux-ci s'avèrent être une structure très efficace pour la manipulation de fonctions booléennes.

Voici quelques propriétés des opérations sur les BDDs :

test d'équivalence en temps constant : du fait de la canonicité, les structures peuvent être stockées de façon unique dans une table et il suffira de comparer l'entrée (pointeur) dans cette table.

négation en temps constant, en inversant le marquage des arcs qui en partent.

connecteurs binaires Les seize connecteurs binaires peuvent s'exprimer en terme de fonction if-then-else $\mathrm{Ite}(x, y, z) = x * y + \bar{x} * z$, par exemple :

$$\begin{aligned} x \wedge y &= \mathrm{Ite}(x, y, 0) \\ x \vee y &= \mathrm{Ite}(x, 1, y) \\ x \oplus y &= \mathrm{Ite}(x, \bar{y}, y) \\ x \Rightarrow y &= \mathrm{Ite}(\bar{x}, 1, y) \end{aligned}$$

quantification existentielle Dire qu'il existe une variable v telle qu'une formule f soit vraie revient à dire f est vraie avec v valant *vrai* ou f est vraie avec v valant *faux* :

$$(\exists v) f \stackrel{déf}{=} f_v \vee f_{\bar{v}}$$

la notation f_v (resp. $f_{\overline{v}}$) indique qu'il s'agit de la formule f dans laquelle la variable v a pour valeur 1 (resp. 0).

quantification universelle le cas symétrique du précédant :

$$(\forall v) f \stackrel{déf}{=} f_v \wedge f_{\overline{v}}$$

substitution Une substitution d'une variable v par une expression w (qui ne contient pas v) peut être calculée ainsi :

$$f[v/w] \stackrel{déf}{=} (\exists v)[(v \equiv w) \wedge f]$$

Le lien entre le μ-calcul présenté à la section 2 et cette représentation symbolique par les BDDs est le suivant. Soit la formule booléenne $R(x, y)$ (cf. 2.1), où R est une variable relationnelle et x et y sont des variables individuelles. Les variables individuelles seront représentées par des vecteurs de variables booléennes, et une variable relationnelle par sa fonction caractéristique sur ces variables (codée par un BDD). On codera (représentera) de la même manière les autres types de formules du μ-calcul.

4. Le module `mu_Calc`

Nous avons choisi de réaliser la maquette en SICStus Prolog. De nombreux modules sont fournis dans l'environnement de programmation standard (graphes, ensembles ordonnés, ...) avec en particulier un module de résolution de contraintes booléennes basé sur les BDD. De plus, le mécanisme des variables attribuées permet de facilement intégrer de nouveaux types de contraintes au niveau de l'unification Prolog.

4.1. *Le module booléen de SICStus*

La plupart des opérations primitives décrites à la section 3 sont implantées dans le module de contraintes booléennes `clpb` de SICStus [Swe96, CAR 94]. Les BDDs sont codés comme des termes Prolog instanciés :

- `$bdd_pos(Var,Fils_Gauche,Fils_Droit)` pour un noeud positif ;
- `$bdd_neg(Var,Fils_Gauche,Fils_Droit)` pour un noeud négatif.

`Var` est ici un index (entier) de variable : les variables sont référencées localement par cet index, et on doit associer un dictionnaire «variables-index» au BDD pour pouvoir déréférencer. On peut noter qu'ici, ce ne sont pas les arcs qui sont marqués (cf. 3), mais les noeuds, le principe restant le même.

- Le test d'équivalence est effectué par le prédicat `$bdd_equiv(BDD1,BDD2)`.
- `$bdd_negate(BDD1,BDD2)` réalise la négation d'un BDD.
- `bdd_univ(BDDx,BDDy,BDDz,BDDres)` calcule récursivement la fonction $\text{Ite}(x, y, z) = res$.
- `$bdd_parts(BDD,Var,FG,FD)` permet de récupérer les composantes d'un noeud.
- Les substitutions sont appliquées par le prédicat `bdd_subst_set` qui donnera le nouveau BDD et le nouveau dictionnaire associé.

Pour implanter la quantification d'une variable v dans une formule $\exists v.f$ ou $\forall v.f$, on calcule le BDD associé à la formule f, on crée un noeud avec une variable d'ordre minimal (*top variable*), on substitue cette variable à v dans le BDD, on récupère les fils gauches et droits, et on leur applique la fonction Ite adéquate (et/ou).

4.2. *Interfaçage avec Prolog*

Le module de variables attribuées de SICStus [Swe96] implante un mécanisme qui permet d'associer des attributs à une variable Prolog. Il autorise le stockage de données relatives à une variable (ici, par exemple, le BDD associé à une variable relationnelle du 2^{e}ordre), mais aussi l'extension de l'algorithme d'unification Prolog à ce nouveau type de variables.

Les relations que l'on définit sont donc des variables Prolog, mais du 2^{e}ordre. La distinction entre variable du 1eret du 2^{e}ordre se fait au niveau de la sémantique.

Les contraintes définissant une relation sont posées à l'aide du prédicat `mu/3` pour les plus petits points fixes et `nu/3` pour les plus grands. Pour calculer la valeur d'une relation, on utilise le prédicat `fp(+Rel,-Params,-Formule)` qui, suivant la définition de `Rel` (avec `mu/3` ou `nu/3`) évaluera le plus petit ou plus grand point fixe. `Params` est une liste de variables représentant les paramètres formels, `Formule` est une formule booléenne représentant la valeur de la relation ainsi évaluée. On pourra aussi dans un deuxième temps énumérer les tuples de booléens composant la relation à l'aide du prédicat `fpa/3`.

La syntaxe (voir section 5 : applications) reprendra celle des expressions booléennes de SICStus, et on utilisera un terme particulier `call(R,Params)` pour modéliser l'«appel» d'un prédicat (relation) `R` avec les arguments `Params` :

Syntaxe SICStus	Formule Booléenne
`X`	x
`0`	*faux*
`1`	*vrai*
`~ F`	$\neg f$
`X ^ F`	$\exists x.f$
`X v F`	$\forall x.f$
`F1 + F2`	$f_1 \vee f_2$
`F1 * F2`	$f_1 \wedge f_2$
`F1 # F2`	$f_1 \oplus f_2$
`F1 =< F2`	$f_1 \Rightarrow f_2$
`F1 =:= F2`	$f_1 \equiv f_2$
`call(R,[X,Y])`	$R(x,y)$

Figure 3. *Syntaxe du module*

Le choix de coder les relations ainsi définies par des variables Prolog fait qu'il n'y a pas persistance de la relation : sa «portée» est celle de la variable. L'utilisateur devra donc faire attention lorsque il définira des relations comportant des appels à d'autres relations.

4.3. *Implantation*

Lorsqu'une contrainte est posée sur une variable relationnelle à l'aide de `mu/3`, on renomme toutes les variables du premier ordre par des termes instanciés de la forme `var(K)` où `K` est un compteur interne à la définition de la relation. On construit ensuite l'arbre syntaxique de la formule en mémorisant au passage les relations qui sont appelées (pour pouvoir par la suite établir le graphe de dépendances des relations). On stocke alors en attribut les termes instanciés correspondants aux paramètres formels (`[var(1),var(2),var(i)]`), l'arbre syntaxique obtenu, les dépendances directes de cette relation et le BDD 0 (*i.e.faux* ou encore $\emptyset$) pour un plus petit point fixe, ou le BDD 1 pour un plus grand point fixe.

Pour calculer la valeur d'une relation, on établit le graphe de ses dépendances, on effectue un tri topologique en composantes connexes et obtient une liste dont chaque élément est une liste de relations mutuellement dépendantes. On lance alors l'évaluation sur chacune de ces composantes connexes («noeud de dépendance»).

Pour chaque «noeud» de dépendance, on va effectuer une itération du calcul

de point fixe sur chacune des relations appartenant à ce «noeud», obtenant alors une liste de BDDs calculés que l'on va comparer aux BDDs obtenus à l'itération précédente. S'il n'y a pas de changement, on a obtenu un point fixe, le calcul pour ce noeud de dépendance s'arrête et on peut passer aux suivants. Sinon, on relance le calcul sur ce noeud, jusqu'à ce que l'on obtienne le point fixe.

Chaque itération sur une relation consiste à calculer récursivement le BDD associé à l'arbre syntaxique : pour un noeud de cet arbre, on va calculer les BDDs associés aux fils de ce noeud, et appliquer la fonction Ite adéquate sur ces BDDs pour obtenir le BDD associé à ce noeud (cf. section 3).

Pour calculer la valeur d'un noeud correspondant à un appel de prédicat (relation), on prend comme valeur le BDD associé à cette relation (qui a déjà été évaluée car le calcul est fait dans l'ordre du tri topologique sur les dépendances), on renomme ses variables dans le contexte courant et on applique la substitution des arguments aux paramètres formels (voir section 3).

Dès qu'une itération d'un calcul sur une relation a été effectuée, on opère une projection du BDD résultant (représentant la valeur actuelle de la relation) sur les paramètres de cette relation et on remplace dans les attributs le BDD associé à cette relation par le nouveau BDD ainsi obtenu. Ce mécanisme continue jusqu'à l'obtention du point fixe (s'il existe).

Enfin, remarquons que la gestion des BDDs n'est pas optimisée. On sait, par exemple, que le coût de la projection d'un BDD sur certaines variables est important. Un mécanisme de cache pour ce type d'opérations sur les BDDs devrait améliorer les performances. De plus, ce module ne permet pas actuellement les définitions imbriquées de points fixes.

5. Applications

5.1. *Analyse d'automates d'états*

On va reprendre ici les exemples développés à partir de l'automate de la figure 1. Voici d'abord la définition de la relation de transition `Edge` :

```
mu(Edge,[X1,X2,X3,Y1,Y2,Y3],
         (((X1=:=0)*(X2=:=0)*(X3=:=1)*(Y1=:=0)*(Y2=:=1)*(Y3=:=0))
        +((X1=:=0)*(X2=:=1)*(X3=:=0)*(Y1=:=0)*(Y2=:=1)*(Y3=:=1))
        +((X1=:=0)*(X2=:=1)*(X3=:=0)*(Y1=:=1)*(Y2=:=0)*(Y3=:=0))
        +((X1=:=0)*(X2=:=1)*(X3=:=0)*(Y1=:=1)*(Y2=:=0)*(Y3=:=1))
        +((X1=:=0)*(X2=:=1)*(X3=:=1)*(Y1=:=0)*(Y2=:=0)*(Y3=:=1))
        +((X1=:=1)*(X2=:=0)*(X3=:=0)*(Y1=:=1)*(Y2=:=0)*(Y3=:=1)))),
fp(Edge,PL,Form).
```

```
PL = [A,B,C,D,E,F],
Form = A*(~B*(~C*(D*(~E*F))))+ ~A*(B*(C*(~D*(~E*F))+
       ~C*(D* ~E+ ~D*(E*F)))+ ~B*(C*(~D*(E* ~F)))) ?
```

L'évaluation de cette relation est immédiate, sa définition n'étant pas récursive. `Form` est la formule booléenne représentant cette relation. Cette formule n'étant pas toujours explicite, on peut ensuite enumérer les tuples booléens composant la relation à l'aide du prédicat `fpa/3`.

```
mu(Edge,[X1,X2,X3,Y1,Y2,Y3],
         (((X1=:=0)*(X2=:=0)*(X3=:=1)*(Y1=:=0)*(Y2=:=1)*(Y3=:=0))
        +((X1=:=0)*(X2=:=1)*(X3=:=0)*(Y1=:=0)*(Y2=:=1)*(Y3=:=1))
        +((X1=:=0)*(X2=:=1)*(X3=:=0)*(Y1=:=1)*(Y2=:=0)*(Y3=:=0))
        +((X1=:=0)*(X2=:=1)*(X3=:=0)*(Y1=:=1)*(Y2=:=0)*(Y3=:=1))
        +((X1=:=0)*(X2=:=1)*(X3=:=1)*(Y1=:=0)*(Y2=:=0)*(Y3=:=1))
        +((X1=:=1)*(X2=:=0)*(X3=:=0)*(Y1=:=1)*(Y2=:=0)*(Y3=:=1)))),
fp(Edge,PL,Form), fpa(PL,Form,Sol).

PL = [A,B,C,D,E,F],
Sol = [[0,1,0,1,0,0],[0,1,0,1,0,1],[0,1,1,0,0,1],[1,0,0,1,0,1],
       [0,0,1,0,1,0],[0,1,0,0,1,1]],
Form = A*(~B*(~C*(D*(~E*F))))+
      ~A*(B*(C*(~D*(~E*F))+~C*(D* ~E+ ~D*(E*F)))+ ~B*(C*(~D*(E* ~F)))) ?
```

Si on interprète le codage booléen, on a bien les transitions : (2,4) (2,5) (3,1) (4,5) (1,2) (2,3). On voit que le codage sur les booléens peut vite devenir assez fastidieux. La relation `Vertex` définit le codage booléen des états : ne sont pas valides les états <0,0,0> (0), <1,1,0> (6) et <1,1,1> (7). Il est clair que l'extension au domaines finis apporterait un gain de lisibilité.

```
mu(Vertex,[X1,X2,X3], ~((X1=:=0)*(X2=:=0)*(X3=:=0)) *
                      ~((X1=:=1)*(X2=:=1))),
```

Par la suite, nous ne reprendrons pas la définition explicite de `Edge` et de `Vertex`. Pour la fermeture transitive de la relation de transition (cf. exemple 1) on pourra écrire :

```
mu(Vertex ...), mu(Edge ...),
mu(Reach,[X1,X2,X3,Y1,Y2,Y3],
  call(Edge,[X1,X2,X3,Y1,Y2,Y3]) +
  (     call(Reach,[X1,X2,X3,Z1,Z2,Z3])
      * call(Edge,[Z1,Z2,Z3,Y1,Y2,Y3])) ,
fp(Reach,PL,Form), fpa(PL,Form,Sol).

PL = [A,B,C,D,E,F],
Sol = [[0,1,0,1,0,0],[0,1,0,0,0,1],[0,1,0,1,0,1],[1,0,0,1,0,1],
       [0,1,0,0,1,0],[0,1,0,0,1,1],[0,0,1,1,0,0],[0,1,1,1,0,0],
       [0,0,1,0,0,1],[0,0,1,1,0,1],[0,1,1,0,0,1],[0,1,1,1,0,1],
       [0,0,1,0,1,0],[0,1,1,0,1,0],[0,0,1,0,1,1],[0,1,1,0,1,1]],
Form = A*(~B*(~C*(D*(~E*F))))+ ~A*(B*(D* ~E+ ~D*(E+F))+
       ~B*(C*(D* ~E+ ~D*(E+F)))) ?
```

Cette contrainte est la traduction directe de la définition de la fermeture transitive.

Voici le calcul du dead-lock (cf. exemple 3) :

```
... ,
mu(Dead,[X1,X2,X3],  call(Vertex,[X1,X2,X3]) *
  (T1 v T2 v T3 v (
    call(Edge,[X1,X2,X3,T1,T2,T3]) =< call(Dead,[T1,T2,T3])))),
fp(Dead,PL,Form), fpa(PL,Form,Sol).

PL = [A,B,C],
Sol = [[1,0,0],[1,0,1]],
Form = A* ~B ?
```

On peut remarquer l'utilisation de la quantification universelle et de l'implication. Le résultat est la relation {4,5} : ce sont bien les seuls états menant obligatoirement à un état bloqué (l'état 5).

Enfin, voici le calcul de la relation unaire des états ne précédant pas l'état 4 (cf. exemple 4) :

```
nu(NP4,[X1,X2,X3],
  call(Vertex,[X1,X2,X3]) *
  (T1 v T2 v T3 v
       (call(Edge,[X1,X2,X3,T1,T2,T3]) =<
           (((T1#1)+(T2#0)+(T3#0)) * call(NP4,[T1,T2,T3])))),
fp(NP4,PL,Form), fpa(PL,Form,Sol).

PL = [A,B,C],
Sol = [[1,0,0],[1,0,1]],
Form = A* ~B ?
```

On utilise ici un plus grand point fixe. La relation de transition n'étant pas réflexive, 4 et 5 sont bien les seuls états non prédécesseurs de l'état 4.

5.2. *Calcul de conditions de terminaison*

Cet exemple est tiré d'un article de Mesnard [MES 96] sur l'inférence de classe de requêtes gauche-terminant. On cherche à exprimer un critère booléen de terminaison pour un programme logique, tel que si l'approximation booléenne d'une requête implique ce critère, l'arbre de calcul Prolog avec résolution standard pour cette requête est fini. On travaille d'abord sur une approximation entière de ces programmes, à partir de laquelle on infère des relations inter-arguments et d'autres informations de contrôle. On utilise ensuite ces informations et l'approximation booléenne du programme afin d'établir un terme booléen qui sera la contition de terminaison. Cette condition étant définie par un système d'équation du μ-calcul propositionnel, on peut facilement la calculer avec notre module Prolog.

On cherche à analyser le petit programme suivant :

```
som3([],B,B).
som3([A1|A],[],[A1|A]).
```

```
som3([A1|A],[B1|B],[A1+B1|C]):-
        som3(A,B,C).

som41(A,B,C,D):-
        som3(A,B,E), som3(E,C,D).
```

pour lequel on est amené à calculer le critère de terminaison `TcSom4` défini par le système suivant :

```
mu(Som3,[A,B,C],
        ((A=:=1)*(B=:=C))+((B=:=1)*(A=:=C)) + call(Som3,[A,B,C])),
mu(MesSom3,[A,B,C], (A=:=1)+(B=:=1)+(C=:=1)),
mu(TcSom41,[A,B,C,D],   E v (call(MesSom3,[A,B,E]) *
        (call(Som3,[A,B,E]) =< call(MesSom3,[E,C,D])))),
fp(TcSom41,PL,Form).

PL = [A,B,C,D],
Form = A*(B+(C+D))+ ~A*(B*(C+D)) ?
```

L'arbre de dérivation Prolog pour la requête $\leftarrow c \wedge som_{41}(A,B,C,D)$ est donc fini si A et B sont deux listes de longueur bornée dans c ou si l'une des deux listes A et B est de longueur bornée dans c et l'une des deux listes C et D l'est aussi.

5.3. *Analyse de clôture*

L'analyse de clôture est l'un des types d'analyses le plus important pour les programmes logiques. L'analyse est réalisée en passant à une approximation du programme logique dans le domaine *Prop*.

Voici comment est réalisée l'analyse de clôture avec notre module. Le programme logique est transformé en sa version booléenne. Nous associons ensuite à chaque prédicat du programme une relation définie comme plus petit point fixe de la disjonction des formules associées à chacune des clauses définissant ce prédicat. Pour obtenir le plus petit modèle booléen du programme, il ne reste alors qu'à évaluer ces relations.

Nous avons comparé cette approche avec une implantation efficace de ce type d'analyse, basée sur un algorithme de Codish [COD 95], calculant le point fixe de l'opérateur T_P.

Voici les résultats obtenus pour l'analyse de quelques programmes bien connus issus de la littérature. Le code de ces benchs est disponible à l'URL suivante :

```
<http ://www.cs.huji.ac.il/ naomil>
```

Langage : Sicstus prolog 3.5 - compiled code
Machine : ULTRA 1 SUN Station
Frequence : 167MHz
RAM : 128Mo
Unite de temps : ms

Nom	#ClP	#VarP	Trans	Scc	#ClB	#VarB	Bu	Mu
qsortapp	15	43	30	0	15	72	60	20
zebra	19	44	30	0	19	81	200	65
money	16	59	30	0	16	96	200	100
grammar	18	44	15	0	18	78	25	0
progeom	20	68	30	10	20	110	110	50
bid	50	111	50	0	50	196	130	50
credit	58	105	50	0	58	218	110	45
warplan	102	242	110	10	109	415	230	90
peephole	135	379	290	10	159	783	690	37
tictactoe	72	515	150	0	74	714	240	50
qplan	150	471	210	15	152	841	1060	3000

Nom : le nom du programme,
#ClP : nombre de clauses du programme initial,
#VarP : nombre de variables du programme initial,
Trans : temps passe a traduire le programme initial dans sa version booleenne,
Scc : temps passe a construire, reduire et trier le graphe d'appel,
#ClB : nombre de clauses du programme booleen,
#VarB : nombre de variables du programme booleen,
Bu : temps de calcul du modele booleen par iteration ascendante,
Mu : idem avec les contraintes du module mu_Calc.

Dans l'ensemble les temps de calculs obtenus avec notre module sont assez bons, excepté pour le programme qplan (dernière ligne). La traduction du programme booléen en formules adéquates du μ-calcul n'est pas optimisée : si le nombre de clauses et de variables du programme booléen est grand, les formules générées seront grosses, et la taille des BDDs qui seront manipulés par la suite explosera, grévant les performances. De plus, comme vu précédemment, la gestion des BDDs à l'intérieur du module n'est pas non plus optimisée.

6. Conclusion

Nous proposons donc un module implantant le μ-calcul sur les booléens à l'aide des diagrammes binaires de décision et s'intégrant à l'environnement SICStus Prolog. Ce module est disponible à l'url : `http ://www.univ-reunion.fr/~gcc`.

Les contraintes sur les relations permises par ce modules offrent une bonne souplesse de programmation - on peut traiter une grande variété de problèmes - ainsi qu'une bonne efficacité comme le montrent les temps obtenus pour l'analyse de clôture sur des programmes logiques non triviaux. Le module doit être amélioré, notamment en optimisant la gestion des BDDs, en permettant les définitions imbriquées de points fixes et en utilisant les domaines finis, ce qui améliorerait la lisibilité. Cet outil est donc, quoique plus souple et s'intégrant bien à SICStus, moins complet et performant que Toupie. Il pourrait aussi être intéressant d'implanter ce module à l'aide des Diagrammes d'expression booléennes (BED) [AND 97] qui semblent mieux adaptés aux calculs de point fixe sur les booléens.

Une ouverture semblant prometteuse est la recherche de nouveaux domaines et des critères nécessaires à la validité des calculs de points fixes dans ces domaines. On sait déjà [COR 97] que l'on peut étendre le μ-calcul aux domaines finis. D'autres domaines pour lesquels des calculs de points fixes sont utiles existent : bases de données, langages rationnels, etc. Ce travail pourrait donc servir de base a un environnement d'aide à l'analyse mettant en jeu divers calculs de points fixes. Enfin, on pourrait aussi s'intéresser aux effets sémantiques de l'introduction de telles contraintes (relations définies par des systèmes de points fixes) dans les langages de programmation logique par contraintes.

Références

[AND 97] Henrik Reif Andersen et Henrik Hulgaard. Boolean expression diagrams. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 88–98, Warsaw, Poland, June 29–July 2 1997. IEEE Computer Society.

[BRY 86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, pages 1035–1044, 1986.

[BRY 92] R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, pages 293–318, 1992.

[BUR 92] J.R. Burch, E.M. Clarke, K.L. MCMillan, D.L. Dill et L.J. Hwang. Symbolic model checking : 10^{20} states and beyond. *Information and Computation*, pages 142–170, 1992.

[CAR 94] M. Carlsson. Boolean constraints in sicstus prolog. Technical Report T91 :09, Swedish Institute of Computer Science, 1994.

[COD 95] M. Codish et B. Demoen. Analizing logic programs using prop-ositional logic programs and a magic wand. *Journal of Logic Programming*, pages 249–274, 1995.

[COL 90] A. Colmerauer. An introduction to Prolog III. *CACM*, 33 (7) :70–90, July 1990.

[COR 97] M-M. Corsini et A. Rauzy. Toupie : The μ-calculus over finite domains as a constraint language. *Journal of Automated Reasoning*, pages 143–171, 1997.

[DIN 88] M. Dincbas, P. Van Hentenrick, H. Simonis, A. Aggoun, T. Graf et F. Berthier. The constraint logic programming language CHIP. *Proc. of FGCS'88*, pages 693–702, 1988.

[JAF 87] J. Jaffar, S. Michaylov, P.J. Stuckey et R.H.C. Yap. The clp(r) language and system. In *Proc. of the ICLP'87*. MIT Press, 1987.

[MCM 92] K.L. McMillan. *Symbolic Model Checking : An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[MES 96] F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. *Proc. of JICSLP'96*, pages 7–21, 1996.

[PAR 74] D. Park. Finiteness is mu-ineffable. In *Theory of Computation Report No. 3*. The University of Warwick, 1974.

[Swe96] Swedish Institute of Computer Science. *SICStus Prolog User's Manual v3.5*, 1996.

[TAR 55] A. Tarski. A lattice-theorical fixpoint theorem and its applications. *Pacific J. Math.*, pages 285–309, 1955.