Preuve de propriétés d'un analyseur syntaxique de plans de missions robotiques

Laurent NANA, François MONIN, Sophie GIRE

Lab-STICC (UMR CNRS 6285), Université de Brest, COMUE UBL
20 avenue Victor Le Gorgeu, BP 817 – CS 93837, 29238 Brest Cedex 3
prenom.nom@univ-brest.fr

RESUME. Afin de renforcer la sûreté de fonctionnement de missions robotiques conçues à l'aide du langage PILOT, un analyseur syntaxique incrémental a été mis en œuvre. Nous avons montré, à l'aide de l'outil SWI-Prolog, que cet analyseur permet de construire tous et uniquement tous les plans syntaxiquement corrects en dessous d'une certaine taille. Cette preuve n'a pu être faite que pour des plans d'une taille inférieure à un certain seuil, en raison d'un problème d'explosion combinatoire lié au fonctionnement de Prolog et à l'approche utilisée. Afin de montrer la validité de l'analyseur incrémental pour tout plan, sans contrainte de taille, nous nous sommes orientés vers l'utilisation d'outils de preuve, et plus particulièrement de l'outil PVS. Dans cet article, nous abordons la modélisation et la vérification de plans PILOT et de leurs propriétés à l'aide de PVS, en vue de la preuve de propriétés susmentionnées de l'analyseur syntaxique incrémental de PILOT.

ABSTRACT. In order to enhance the dependability of robotics missions designed with the help of the language PILOT, an incremental syntactic analyzer has been built. We have shown, with the help of the SWI-Prolg tool, that the analyzer allows to build all and only all the plans which are syntactically correct under some size. This proof has been done only for plans whose size is less than a threshold, because of a combinatory explosion problem inherent to the working of Prolog and to the used approach. In order to show the validity of the incremental syntactic analyzer for all plans, without any size constraint, we turned to the use of proof-based approaches, and particularly towards PVS tool. This paper deals with the modeling and verification of PILOT plans and their properties with the help of PVS, in order to prove the above properties of the incremental syntactic analyzer of PILOT.

MOTS-CLES : programmation de missions, robotique, modélisation, vérification, preuves

KEYWORDS: missions programming, robotics, modeling, verification, proof

1. Introduction

Le travail présenté dans cet article se situe dans le prolongement de travaux antérieurs ayant pour but de renforcer la sûreté de fonctionnement et la sécurité d'applications robotiques (Barbier *et al.*, 2006 ; Barrouil et Lemaire, 1999 ; Py et Ingrand, 2004 ; Rutten, 2000 ; Turro, 1999 ; Nana, 2007 ; Nana *et al.*, 2005a, 2005b, 2003 ; Nana Tchamnda *et al.*, 2002 ; Laouamer *et al.*, 2012). Il vise l'utilisation d'outils de preuves et plus particulièrement de l'outil PVS, pour prouver des propriétés d'un analyseur syntaxique incrémental de plans de missions conçus à l'aide du langage PILOT, un langage de programmation de missions conçu et mis en œuvre au sein du Lab-STICC. Un travail antérieur basé sur l'utilisation de Prolog et de l'outil SWI-Prolog développé à l'Université d'Amsterdam nous a permis de prouver que l'analyseur syntaxique incrémental permet de construire tous et uniquement tous les plans syntaxiquement corrects. Toutefois, cette preuve n'a pu être faite que pour des plans d'une taille inférieure à un certain seuil, en raison d'un problème d'explosion combinatoire lié au fonctionnement de Prolog et à l'approche utilisée. L'utilisation d'outils de preuves permet de s'affranchir de la contrainte de taille rencontrée dans cette approche. Le choix de PVS est en partie dû à notre expérience dans l'utilisation de cet outil pour la vérification de protocoles (Groote *et al.*, 2004; 1998). L'utilisation d'outils de preuve nécessite une formalisation adéquate de la syntaxe du langage PILOT et du fonctionnement de l'analyseur syntaxique incrémental. Cet article se focalise sur la modélisation de plans de missions, de leurs propriétés syntaxiques, et d'opérations liées à la construction incrémentale de plans, en vue de la preuve de propriétés envisagée.

Dans la deuxième section, nous présentons succinctement le langage PILOT et l'approche de construction et d'analyse syntaxique incrémentale mise en œuvre pour la construction de plans PILOT. La troisième section est consacrée à la preuve de propriétés faite à l'aide de SWI-Prolog. Dans la quatrième section, nous montrons l'inadéquation (par rapport à PVS) des modèles élaborés dans l'approche précédente et présentons les modèles proposés pour la preuve de propriétés de plans de missions et d'opérations de construction de plans à l'aide du formalisme de PVS. L'article se termine par un bilan des travaux effectués.

2. Le langage PILOT et l'approche de construction et de vérification syntaxique incrémentale de plans

2.1. Le Langage PILOT

Le langage PILOT (Le Rest, 1996 ; Fleureau, 1998) est basé sur la notion d'action. Une action comporte un ordre exécutable par le robot, une précondition et une ou plusieurs règles de surveillances auxquelles sont associés des traitements. Deux types d'actions sont distingués : les actions élémentaires et les actions continues. Une action élémentaire a sa propre fin, contrairement à une action continue. Elle se termine généralement lorsque son objectif est atteint. La terminaison d'une action continue est quant à elle déclenchée par une autre primitive du langage. Quelle que soit sa nature, élémentaire ou continue, une action ne s'exécute que si sa précondition est vraie. De même, lorsqu'au cours de l'exécution d'une action, une de ses règles de surveillance est vraie, le traitement associé est exécuté. Dans la pratique, les préconditions et les règles de surveillance associées aux actions sont très souvent des conditions sur des valeurs de capteurs. Le traitement par défaut associé aux règles de surveillances est l'arrêt de l'action correspondante.

Le langage PILOT comporte des structures de contrôle permettant la construction de plans : séquentialité, conditionnelle, itération, parallélisme et préemption. Nous les présentons ci-après :

— La séquentialité : elle est marquée par un début de séquence et une fin de séquence. Elle permet la mise en séquence des différentes autres primitives offertes par le langage (actions et structures de contrôle).

— La conditionnelle : elle est formée d'une ou plusieurs alternatives ordonnées du haut vers le bas et comportant chacune une condition suivie d'une séquence. La première séquence dont la condition est vraie est la seule exécutée.

— L'itération : elle est formée d'un critère de poursuite suivi d'une séquence. Ce critère peut être soit un nombre d'itérations, soit une expression booléenne. Dans le premier cas, l'itération est dite fixe et dans le second, elle est qualifiée de conditionnelle.

— Le parallélisme : il est formé de plusieurs séquences. Les séquences sont exécutées en parallèle. Son exécution se termine lorsque toutes les séquences ont atteint leur fin.

— La préemption : comme le parallélisme, elle est formée de plusieurs séquences dont l'exécution se fait en parallèle, mais contrairement à ce dernier, son exécution se termine dès que l'une de ses séquences atteint sa fin.

Après cette brève présentation du langage PILOT, nous présentons l'approche de construction incrémentale de plans de missions dans la section suivante.

2.2. Approche de construction et vérification syntaxique incrémentale de plans PILOT

L'approche de construction et vérification incrémentale de plans garantit la validité syntaxique du plan à tout moment. Quand le programmeur d'applications PILOT commence la construction d'un plan, il a une séquence vide. Il peut ensuite utiliser les opérations *insérer*, *effacer* et *modifier* pour continuer la construction de son plan. L'opération *modifier* permet par exemple de changer le nombre de boucles d'une itération, une condition dans une structure conditionnelle, etc. Chaque fois qu'une opération est appliquée, le système vérifie la validité syntaxique du plan résultant. La mise à jour n'est réellement effectuée que si le plan résultant est syntaxiquement correct. A défaut, le plan reste inchangé et un message d'erreur est envoyé au programmeur. Afin d'assurer la validité après chaque insertion, des structures par défaut ont été définies pour les différentes structures du langage.

3. ***Preuve de propriétés de l'approche de construction incrémentale de plans PILOT à l'aide de SWI-Prolog***

Il s'agit ici de montrer que le nouveau mécanisme de vérification syntaxique de plans mis en œuvre permet de ne construire que des plans syntaxiquement corrects. Pour des raisons de simplification, nous nous limitons aux plans construits uniquement à l'aide d'opérations d'insertion.

Le réseau de Petri coloré de la figure 1 illustre l'approche utilisée pour la «validation».

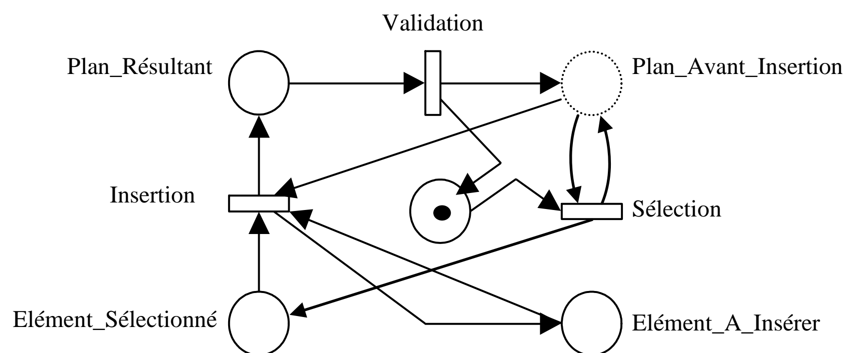

Figure 1. L'approche de validation

Le marquage initial de la place Plan_Avant_Insertion est {*[debut_sequence*, *fin_sequence]*} et celui de la place Elément_A_Insérer est {*debut_sequence*, *fin_sequence*, *action_continue*, *action_elementaire*, *parallelisme*, *preemption*, *iteration*, *conditionnelle*, *expression_booleenne*, *nombre_de_boucles*}. Les autres places (Elément_Selectionné et Plan_Résultant) n'ont aucun marquage initial. Les transitions Sélection et Insertion modélisent respectivement la sélection d'un élément du plan et l'insertion d'un élément dans le plan. La transition Validation représente l'analyseur syntaxique formel du langage PILOT. Le nouveau mécanisme de vérification syntaxique mis en œuvre est sans erreur si ce réseau de Petri est sans blocage.

Afin d'appliquer cette approche de « validation » à la mise en œuvre du mécanisme de construction et de vérification syntaxique, un analyseur syntaxique a été construit en PROLOG en utilisant l'approche proposée par Giannesini et al (Giannesini *et al.*, 1985), puis la construction de plans a été modélisée et finalement, les propriétés suivantes ont été évaluées sur le modèle :

— Une insertion pourrait-elle conduire à un plan syntaxiquement incorrect ?

— Y a-t-il un plan syntaxiquement correct qui ne puisse être construit à l'aide du modèle d'insertion ?

La première propriété permet de s'assurer que le mécanisme de vérification syntaxique permet à l'utilisateur de ne construire que des plans syntaxiquement corrects. Le mécanisme pourrait également être très restrictif, le risque étant qu'il interdise une opération d'insertion qui conduit à un plan syntaxiquement correct. La seconde propriété permet d'éviter un tel désagrément.

Compte tenu du fonctionnement de PROLOG, il est nécessaire de considérer un ensemble borné de plans. Autrement, PROLOG essaiera de calculer tous les plans syntaxiquement corrects, ce qui conduira à des problèmes de mémoire. Pour cette raison, l'analyseur syntaxique PROLOG (également utilisé comme synthétiseur de plans syntaxiquement corrects) et le modèle de construction de plans PILOT ont été réécrits pour ne calculer que l'ensemble des plans dont la longueur est inférieure à une valeur passée en paramètre. La longueur du plan est le nombre d'éléments de la liste modélisant le plan. Avec ces nouvelles contraintes, la traduction en PROLOG des propriétés ci-dessus est la suivante (figure 2):

```
probleme_insertion (Taille_Max, Plan, Plan_Graphique) :-
        insertions_automatiques (Taille_Max, Liste_Plans),
        member (Plan_Graphique, Liste_Plans),
        conversion (Plan_Graphique, Plan),
            not syntaxiquement_correct (Plan,[]).
rejet_anormal (Taille_Max, Plan):-
                insertions_automatiques (Taille_Max, Ensemble_Plans_Graphiques),
                conversion_ensemble (Ensemble_Plans_Graphiques, Ensemble_Plans),
                plans_syntaxiquement_corrects (Taille_Max,Ensemble_Plans_Valides),
                member (Plan, Ensemble_Plans_Valides),
            not member (Plan, Ensemble_Plans).
```

Figure 2. Traduction en Prolog des propriétés de validité de l'analyseur

L'opération **conversion** (**conversion_ensemble**) transforme un (ensemble de) modèle(s) graphique(s) en un (ensemble de) modèle(s) qui est (sont) utilisable(s) par l'analyseur syntaxique PROLOG.

Pour la programmation, nous avons utilisé l'outil SWI-Prolog développé à l'Université d'Amsterdam. Les tests effectués pour des plans de taille inférieure ou égale à 14, n'ont révélé aucun problème d'insertion et aucun rejet anormal. On peut donc conclure, pour des plans construits par insertion et de taille inférieure ou égale à 14, que le mécanisme de construction et de vérification syntaxique incrémentale proposé et mis en œuvre dans l'environnement de contrôle PILOT permet de construire tous et uniquement tous les plans syntaxiquement corrects. Une exception « mémoire insuffisante » est levée pour des plans de taille supérieure à 14. Cette exception est due à une explosion du nombre de cas à traiter.

Dans la section qui suit, nous présentons la modélisation et la vérification de propriétés de plans PILOT et d'opérations sur les plans, à l'aide de PVS (Prototype Verification System). PVS (Owre *et al*, 2001 ; Shankar *et al*, 2001) offre une approche de preuve formelle permettant de s'affranchir du problème d'explosion combinatoire susmentionné.

4. Modélisation de plans et d'opérations pour la preuve de propriétés à l'aide de PVS

4.1. Analyse du modèle adopté dans l'approche SWI-Prolog

Dans l'approche de modélisation adoptée pour la preuve de propriétés de l'éditeur incrémental de plans PILOT à l'aide de Prolog et de l'outil SWI-Prolog, le plan a été représenté comme une liste d'éléments numérotés afin de refléter le plan construit à l'aide de l'éditeur graphique, et de permettre d'identifier de façon précise la localisation des éléments du plan. En effet, pour une opération telle que l'insertion, l'on a besoin d'identifier le point d'insertion. Dans l'interface graphique de PILOT, une insertion se fait en sélectionnant l'élément à insérer dans une liste prédéfinie d'éléments, puis en cliquant sur l'élément avant lequel l'insertion doit être faite. La numérotation adoptée dans l'approche utilisée sous Prolog convenait bien compte tenu de la démarche consistant, à partir d'un plan donné, à générer l'ensemble des plans possibles que l'on peut construire par insertion d'un élément unique (structure par défaut) quelconque. Cette numérotation pose toutefois un problème de généralisation dans le cadre de l'approche de preuve formelle envisagée avec PVS, étant donné que l'on doit raisonner sur tout plan quelle que soit sa taille. L'affectation de numéro devient donc un frein à la généralisation. L'une des difficultés a donc été la proposition d'un modèle de plan non basé sur l'utilisation de numéros, et permettant d'identifier de façon unique les éléments du plan, dans les différentes opérations liées à la construction et la vérification syntaxique incrémentales de plans.

Les opérations de construction et de vérification syntaxique incrémentale nécessitent de pouvoir identifier pour un élément du plan l'englobant correspondant. Dans l'outil SWI-Prolog, un mécanisme permettant d'isoler des blocs par une notation du type « par : numéro:Liste » est disponible et facilite la localisation des englobants. La localisation des englobants est également essentielle pour la mise en œuvre du processus de preuve dans PVS, mais ce dernier n'offre aucun mécanisme semblable à celui de SWI-Prolog. Il est par conséquent nécessaire de proposer une solution différente pour la localisation et l'identification des englobants d'éléments du plan.

4.2. Proposition de modèles pour PVS

La preuve des propriétés de l'analyseur syntaxique incrémental de PILOT nécessite la définition préalable de plusieurs modèles sous PVS, dont celui d'un plan syntaxiquement valide, celui de l'englobant d'une sélection, celui décrivant l'opération de sélection d'un élément du plan avant lequel l'insertion est souhaitée. Nous présentons ci-après les modèles proposés pour la représentation de ces différentes entités sous PVS.

4.2.1. Définition d'un modèle de plan

Le plan est modélisé par une liste d'éléments suivant l'approche utilisée en langage Lisp. L'environnement PVS utilise en effet des structures proches de celles de Lisp. L'ensemble des éléments utilisés pour la représentation du plan est {*ds, fs, e, c, par, pre, exp, condi, iter*}, où ***ds*** désigne le début de séquence, ***fs*** la fin de séquence, ***e*** une action élémentaire, ***c*** une action continue, ***par*** une primitive parallèle, ***pre*** une primitive préemption, ***exp*** une expression conditionnelle, ***condi*** une primitive conditionnelle et ***iter*** une primitive itération. Ces éléments correspondent aux terminaux que l'on retrouve dans la syntaxe « ajustée » du langage PILOT.

La figure 3 illustre la modélisation du plan à travers quelques exemples :

Plan PILOT	**Modèle PVS**
Séquence vide	`cons (ds, cons (fs, null))`
Plan formé d'une action élémentaire	`cons (ds, cons (e, cons (fs, null)))`
Plan formé d'une primitive itération	`cons (ds, cons (iter, cons (exp, cons (ds, cons (e, cons (fs, cons (fs, null)))))))`

Figure 3. Illustration de la modélisation du plan en PVS

4.2.2. Définition des propriétés caractérisant un plan syntaxiquement correct

Le principe général consiste, étant donné l'ensemble des règles définissant la grammaire du langage, à représenter chaque règle par une fonction qui prend en entrée une liste d'éléments à analyser, ainsi que le reste attendu après reconnaissance du motif satisfaisant la règle (suivant une analyse de gauche à droite), et retourne vrai s'il y a concordance entre le reste attendu et le reste effectif après reconnaissance du motif. L'application de ce principe permet, dans le cas de Prolog, de générer automatiquement et sans difficulté le programme Prolog reconnaissant le langage, comme le montre le tableau 1 donnant le traitement des premières règles de la grammaire du langage PILOT :

Règles syntaxiques de PILOT		Représentation en prolog
S	: S_BASE	`plan_valide (U,V) :- s_base (U,V).`
S_BASE	: ds L_PRIM_BASE fs	`s_base ([ds|U],V) :- l_prim_base (U, [fs,V]).`
L_PRIM_BASE	: ε |	`l_prim_base (U,U).`
	PRIM_BASE L_PRIM_BASE	`l_prim_base (U,V) :- prim_base (U,W), l_prim_base (W,V).`
PRIM_BASE	: PRIM_PARALLELE |	`prim_base (U,V) :- Prim_parallele (U,V).`
	PRIM_PREEMPTION |	`prim_base (U,V) :- Prim_preemption (U,V).`
	PRIM_CONDITIONNELLE |	`prim_base (U,V) :- Prim_conditionnelle (U,V).`
	PRIM_ITERATION |	`prim_base (U,V) :- Prim_iteration (U,V).`
	PRIM_ACT_ELEM	`prim_base (U,V) :- Prim_act_elem (U,V).`
….	…	…

Tableau 1. Traduction en Prolog des règles de la syntaxe du langage PILOT

Avec le même principe, sous PVS, la représentation a la forme suivante (figure 4) :

```
1. % définition de types et déclaration de variables
2. prim : type = {ds, fs, …}
3. t : type = list[prim]
4. u, v, w, x, y : t
5. % représentation des règles
6. plan_valide (u,v) : bool = s_base (u,v)
7. s_base (u,v) : bool = (car(u) = ds) and l_prim_base (cdr(u),cons(fs,v))
8. l_prim_base (u,v): recursive bool = ((u=v) or (u /= v and u/=null and exists w: (prim_base(u,w)
      and l_prim_base (w,v)))) measure lambda (x,y):length (x)
9. prim_base(u,v) : bool = prim_parallele(u,v) or prim_preemption (u,v) or prim_conditionnelle (u,v)
      or prim_iteration (u,v) or prim_act_elem (u,v)
```

Figure 4. Syntaxe de PILOT sous PVS suivant l'approche adoptée pour Prolog

Cette représentation n'est malheureusement pas utilisable telle quelle sous PVS, car le langage de description associé à PVS ne permet d'utiliser une fonction qu'après l'avoir définie. Une solution aurait été de faire au préalable une définition incomplète des fonctions (spécification des fonctions), et de les définir entièrement ensuite, mais le langage de PVS ne le permet pas. D'autre part, la nature récursive de la syntaxe, et plus particulièrement la récursivité croisée qu'elle incorpore, ne permet pas d'obtenir un schéma de dépendance complet à partir duquel l'on puisse définir les fonctions les unes après les autres suivant l'ordre de dépendance.

Pour s'affranchir de ces inconvénients, une première approche a consisté à élaborer le schéma de dépendance des règles et à définir dans l'ordre de dépendance des fonctions, celles pour lesquelles le problème de récursivité croisée ne se pose pas, puis à utiliser une approche de passage de fonctions en paramètre pour la définition de fonctions liées par des appels récursifs croisés. Le principe est illustré à travers l'exemple suivant des fonctions de parité et d'imparité.

Pair (n : N) : booléen = n=0 ou (n ≠0 et impair (n-1))
Impair (n : N) = n=1 ou (n > 1 et pair (n-1))

Cette définition des fonctions pair et impair peut être transformée comme suit (figure 5), afin de résoudre le problème de récursivité croisée qui se pose sous PVS :

```
1.   Pair_1 (n: N, imp: N -> booléen): booléen = n = 0 ou (n /= 0 et imp (n-1))
2.   Impair (n : N) : booléen = n=1 ou (n > 1 et Pair_1 (n-1, Impair))
3.   Pair (n : N) : booléen = Pair_1 (n, Impair)
```

Figure 5. Représentation des fonctions pair et impair pour PVS avec "résolution" des appels récursifs croisés

Cet exemple d'appels récursifs croisés reste toutefois plus simple à traiter comparativement aux situations rencontrées dans la syntaxe du langage PILOT, qui mettent en jeu plus de deux références croisées. C'est par exemple le cas de la règle « s_base » qui fait référence à « l_prim_base » qui fait elle-même référence à « prim_base » qui fait à son tour référence à «prim_parallelisme» qui fait également référence à « s_base ».

D'autre part, la définition d'une fonction récursive sous PVS requiert de lui associer une fonction « mesure » décroissante au fil des appels récursifs et ayant une borne inférieure, afin de garantir la terminaison des appels récursifs. Dans le cas des fonctions *pair* et *impair* définies ci-dessus, il suffit d'associer comme fonction « mesure », la fonction identité. La définition de fonctions de mesures appropriées induit des contraintes dans la représentation de la syntaxe du langage PILOT sous PVS. L'on trouvera ci-après (figure 6) un extrait simplifié de la représentation proposée, illustrant le traitement adopté pour l'exemple susmentionné de récursivités croisées multiples présentes dans la syntaxe de PILOT:

```
1. prim_parallele_i (u,v,lsq):bool = (u/= null and v /= null and car (u) = par and lsq (cdr(u),v))
2. prim_base_i(u,v,lsq):bool = prim_act_elem (u,v) or prim_parallele_i(u,v,lsq) or …
3. l_prim_base_i (lsq)(u,v): recursive bool = (u=v) or (u /= v and exists w: (prim_base_i (u,w,lsq)
      and l_prim_base_i (lsq)(w,v)) measure lambda (lsqf)(x,y):length (x)
4. s_specifique (u,v): bool = u = cons(ds,cons(c,cons(fs,v)))
5. s_base_i (u,v,lsq): bool = u /= null and car(u) = ds and l_prim_base_i (lsq)(cdr(u),cons(fs,v))
6. s_i (u,v,lsq) : bool = s_base_i (u,v,lsq) or s_specifique (u,v)
7. l_seq_qcq (u,v): recursive bool = u=v or (u/=v and exists w: and s_i (u,w,l_seq_qcq) and
      l_seq_qcq(w,v))
      measure lambda (x,y):length (x)
8. l_seq (u,v):bool = exists w, z: (l_seq_qcq(u,w) and s_base_i (w, z,l_seq_qcq) and l_seq_qcq(z,
      v))
9. s_base (u,v): bool = s_base_i(u,v,l_seq)
10.  plan_valide(u): bool = s_base (u,null)
```

Figure 6. Extrait de la représentation de la syntaxe de PILOT sous PVS avec "résolution" des appels récursifs croisés

Cette approche a permis de générer une représentation de la syntaxe de PILOT sous PVS. Afin de la valider, les obligations de preuves générées par PVS ont été prouvées et quelques théorèmes définissant des plans ont été définis et prouvés à l'aide de PVS, dont les théorèmes de la figure 7:

```
1.  p1 : theorem plan_valide (cons(ds,cons(fs,null)))
2.  p2 : theorem plan_valide (cons(ds,cons(e, cons(fs,null))))
3.  p3 : theorem plan_valide (cons(ds,cons(e,cons(e, cons(fs,null)))))
4.  p4 : theorem plan_valide (cons(ds,cons(par,cons(ds,cons(fs,cons(fs,null))))))
5.  p5 : theorem plan_valide (cons(ds,cons(par,cons(ds,cons(fs,cons (e, cons(fs,null)))))))
6.  p6 : theorem plan_valide (cons(ds,cons(par,cons(ds,cons (e,cons(fs,cons (ds,cons (fs,
      cons(fs,null)))))))))
7.  p7 : theorem plan_valide (cons(ds,cons(condi,cons (exp, cons(ds,cons(fs,cons(fs,null)))))))
8.  p8 : theorem plan_valide (cons(ds,cons(e,cons(condi,cons (exp, cons(ds,cons(e,cons(fs,cons
      (e,cons(fs,null))))))))))
9.  p9  : theorem plan_valide (cons(ds, cons(iter, cons (exp, cons(ds, cons(fs, cons(fs,null)))))))
10. p10: theorem plan_valide (cons (ds, cons(iter, cons (exp, cons(ds, cons (e, cons(fs, cons (e,
      cons(fs,null)))))))))
```

Figure 7. Exemples de théorèmes prouvés à l'aide de PVS pour la validation de la représentation de la syntaxe de PILOT sous PVS

4.2.3. Modélisation de la sélection d'un élément du plan

Il s'agit de modéliser le point d'insertion tel que défini dans PILOT. Dans PILOT, ce point d'insertion est caractérisé par deux éléments : l'élément du plan sur lequel l'utilisateur a cliqué, et la primitive dite englobante contenant cet élément. Le modèle proposé pour la représentation de l'élément sélectionné est le couple (Lav, Lap) où Lav représente la partie du plan le précédant et Lap l'autre moitié l'incluant. Ce choix nous permet de nous appuyer sur les propriétés de la fusion de listes pour faciliter la modélisation des opérations de sélection, mais aussi, d'être en cohérence avec le modèle de représentation adopté dans la modélisation des règles syntaxiques (utilité du reste attendu après identification du motif satisfaisant la règle). Dans ce modèle, la primitive englobante se trouve très souvent répartie entre les 2 moitiés du plan Lav et Lap. La figure 8 illustre le modèle adopté pour la sélection d'un élément du plan P. Dans ce schéma, Leng est la liste partant de l'englobant jusqu'à la fin du plan.

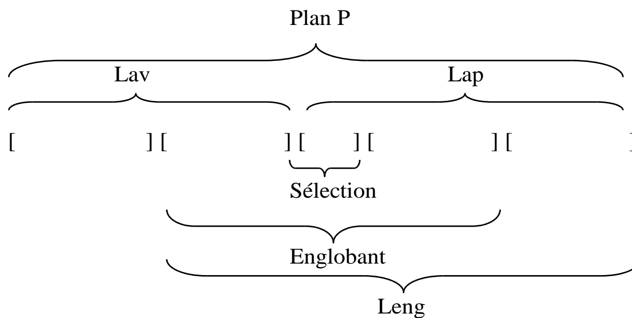

Figure 8. Modèle adopté pour la sélection d'un élément du plan

Nous donnons en figure 9 un extrait de la description en PVS de l'opération de sélection, limité à la sélection d'un élément situé au 1er niveau du plan. Dans ce cas, il n'y a pas d'englobant.

```
selection (P, Lav, Lap, Leng): recursive bool =
     P = fusion (Lav,Lap) and
     (  % selection dans la sequence
        ( Leng = null and ( Lap = cons(fs, null) or
           % sélection du début de séquence
           Lav = null or
           % selection d'un autre élement de la séquence
           (l_prim_base (Lap,cons(fs,null))))) or…
     )
     measure lambda (s,s1,s2,s3):length (s)
```

Figure 9. Extrait de la représentation de l'opération de sélection sous PVS

Conclusion

Le travail présenté dans cet article avait pour but l'utilisation de PVS pour la preuve de propriétés de l'analyseur syntaxique de plans PILOT, afin de s'affranchir d'un problème d'explosion combinatoire rencontré dans un travail antérieur basé sur l'utilisation de SWI-Prolog. Notre objectif était également d'expérimenter l'utilisation de PVS et d'évaluer son adéquation pour la réalisation de preuves relativement complexes impliquant la modélisation d'entités de natures différentes (syntaxe d'un langage, opérations, etc) fréquentes dans les systèmes de contrôle-commande.

Dans ces perspectives, nous avons étudié et proposé une représentation de la syntaxe de PILOT sous PVS, basée sur la représentation des différentes règles syntaxiques de PILOT par des fonctions PVS. Cette représentation a été « validée » par la vérification des obligations de preuves générées par PVS et la preuve de théorèmes définissant des plans PILOT. L'une des principales difficultés rencontrées dans la représentation de la syntaxe de PILOT en PVS a été la modélisation en PVS des appels récursifs croisés. En effet, contrairement à Lisp et à Prolog, le langage de description associé à PVS ne permet d'utiliser une fonction qu'après l'avoir entièrement définie. Nous avons par conséquent proposé une représentation supprimant les récursivités croisées et garantissant la décroissance de la chaîne traitée à chaque appel récursif, afin d'assurer la terminaison de tels appels. Notre modélisation de l'opération de sélection de l'éditeur orienté syntaxe a également été présentée dans cet article. Les résultats obtenus à cette phase du travail permettent de conclure en un bon potentiel de PVS pour la réalisation des preuves envisagées.

Bibliographie

Barbier M., Gabard J. F., Vizcaino D., Bonnet-Torrès O. (2006). ProCoSA: a software package for autonomous systems supervision. *Proceedings of 1st National Workshop on Control Architectures of Robots: software approaches and issues*, Montpellier, France.

Barrouil C., Lemaire J. (1999). Advanced Real-Time Mission Management for an AUV. *Proceedings of SCI NATO RESTRICTED Symposium on Advanced Mission Management and System Integration Technologies for improved Tactical Operations,* Florence, Italy.

Fleureau J.-L. (1998). *Vers une méthodologie de programmation d'un système de télérobotique : comparaison des approches PILOT et Grafcet.* Thèse de doctorat, Université de Rennes 1.

Giannesini F., Kanoui H., Pasero R., Van Caneghem M. (1985). *Prolog.* Paris, InterEditions.

Groote J.F. , Monin F.,Springintveld J. (2004). A computer checked algebraic verification of a distributed summation algorithm. *Formal Aspects of Computing*. Springer-Verlag. Published Online.

Groote J.F., Monin F., Van de Pol J.C. (1998). Checking verifications of protocols and distributed systems by computer. *Proceedings of Concur'98*, Sophia Antipolis, France.

Laouamer L., Benhocine A., Nana L. et Pascu A. (2012). Motion JPEG Video Authentication based on Quantization Matrix Watermarking: Application in Robotics. International Journal of Computer Applications (IJCA), Foundation of Computer Science, New York, USA. Vol. 47, N°24, pp. 1-5.

Le Rest E. (1996). *PILOT : un langage pour la télérobotique.* Thèse de doctorat, Université de Rennes 1.

Nana L. Investigating safety mechanisms for robotics applications. (2007). *IPSI BgD Transactions on Internet Research,* vol. 3, n° 1, pp. 45-50.

Nana L., Marcé L., Opderbecke J., Perrier M., Rigaud V. (2005). Investigation of safety mechanisms for oceanographic AUV missions programming. *Proceedings of the IEEE OCEANS'05 Europe Conference,* Brest, France.

Nana L., Singhoff F., Legrand J., Vareille J., Le Parc P., Monin F., Massé D., Marcé L., Opderbecke J., Perrier M., Rigaud V. (2005). Embedded intelligent supervision and piloting for oceanic AUV. *Proceedings of the IEEE OCEANS'05 Europe Conference*, Brest, France.

Nana L., Legrand J., Singhoff F., Marcé L. (2003). Modelling and Testing of PILOT Plans Interpretation Algorithms. *Proceedings of Multi-conference on Computational Engineering in Systems Applications, CESA'03, IEEE*, Lille, France.

Nana Tchamnda L., Nicolas V-A., Marcé L. (2002). Towards a formal approach for the regeneration of PILOT control system. *Proceedings of 6th World Multiconference on Systemics, Cybernetics and Informatics, SCI'2002, IEEE Venezuela*, Orlando, Florida, USA.

Owre S *et al*. (2001). *PVS System Guide*. Rapp. Tech. SRI International.

Py F., Ingrand F. (2004). Dependable Execution Control for Autonomous Robot. *Proceedings of IROS 2004 (IEEE/RSJ International Conference on Intelligent Robots and Systems)*, Sendai, Japan.

Rutten E. (2000). *A framework for using discrete control synthesis in safe robotic programming*, Rapport de recherche INRIA, 2000.

Shankar N *et al* . (2001). *PVS Prover Guide*. Rapp. Tech. SRI International.

Turro N. (1999). *MaestRo: Une approche formelle pour la programmation d'applications robotiques*. Thèse de doctorat, Université de Nice, Sophia Antipolis.