4e Conférence Francophone de MOdélisation et SIMulation
"Organisation et Conduite d'Activités dans l'Industrie et les Services"
MOSIM'03 – du 23 au 25 avril 2003 - Toulouse (France)

MODELISATION, SIMULATION ET TEST DES ALGORITHMES D'INTERPRETATION DE PLANS PILOT

Laurent NANA, Jerôme LEGRAND, Frank SINGHOFF et Lionel MARCE

Université de Bretagne Occidentale
Département d'Informatique – Projet LIMI, EA2215
20 Av. Victor Le Gorgeu, BP 832
29285 Brest Cedex FRANCE
Mél : {nana,jlegrand,singhoff,marce}@univ-brest.fr

RESUME : *PILOT (Programming and Interpreted Language Of actions for Telerobotics) est un langage graphique et interprété destiné à la télérobotique. Il dispose d'un système de contrôle dont les différents modules ont été modélisés à l'aide d'automates à états finis. Le travail décrit dans cet article fait suite à un travail précédent qui a consisté en l'application de techniques de tests statiques et dynamiques aux programmes de l'interpréteur de plans PILOT (Nana Tchamnda et al., 2002). Le but est d'adopter une démarche plus « formelle » visant à modéliser les algorithmes d'interprétation, à vérifier leur conformité par rapport à la sémantique opérationnelle du langage, à corriger les éventuels dysfonctionnements, puis à re-générer le code de l'interpréteur à partir du modèle validé. Nous utilisons les réseaux de Petri colorés pour la modélisation. L'outil utilisé est Design/CPN (http://www.daimi.aau.dk/DesignCPN). Il a été développé aux Etats-Unis et est maintenu par l'Université de Aarrhus, Danemark.*

MOTS-CLES : *Modélisation, test, simulation, réseaux de Petri, langages, télérobotique.*

1. INTRODUCTION

Le langage PILOT (Le Rest, 1996, Le Rest et al., 1997) est doté d'un système de contrôle dont les différents modules ont été modélisés à l'aide d'automates à états finis (Fleureau, 1998), mais dont la mise en œuvre n'a fait l'objet ni d'une génération automatique à partir des spécifications, ni d'une vérification formelle. Le but du travail décrit dans cet article est d'adopter une démarche plus rigoureuse visant à modéliser les algorithmes d'interprétation, à vérifier leur conformité par rapport à la sémantique opérationnelle du langage, à corriger les éventuels dysfonctionnements, puis à régénérer le code de l'interpréteur à partir du modèle validé. Il se situe dans le prolongement des travaux décrits dans (Nana Tchamnda et Marcé, 2001) et fait suite à un travail précédent qui a consisté en l'application de techniques de tests statiques et dynamiques aux programmes de l'interpréteur de plans PILOT (Nana Tchamnda et al., 2002). En effet, bien que ces techniques se soient révélées très utiles dans la détection et la correction d'erreurs dans les programmes d'interprétation, leur utilisation ne permet pas de garantir la conformité à la sémantique opérationnelle du langage PILOT. Nous utilisons les réseaux de Petri (RdP) colorés pour la modélisation. Le support logiciel est Design/CPN. Cet outil nous permet non seulement de simuler et de vérifier les propriétés de nos modèles, mais aussi d'avoir une représentation modulaire et une expression simple de certains concepts utilisés tels que la lecture non destructive de l'information. Design/CPN a été développé aux Etats-Unis et est maintenu par l'Université de Aarrhus au Danemark.

Nous commencerons cet article par une brève présentation du langage PILOT et de son système de contrôle, avec un accent sur le fonctionnement de l'interpréteur. Nous ferons ensuite un bref aperçu de la modélisation des systèmes à événements discrets et argumenterons le choix des RdP comme support pour la modélisation des algorithmes d'interprétation. Cette partie nous permettra également de situer notre travail par rapport à d'autres travaux voisins. La partie suivante consistera en la présentation de la modélisation du plan, étape indispensable à la simulation de l'exécution des algorithmes d'interprétation. La modélisation des algorithmes d'interprétation sera ensuite examinée. Nous finirons cet article par le test de ces modèles et l'interprétation des résultats.

2. PILOT : UN LANGAGE ET UN SYSTEME DE CONTROLE

2.1. Le langage PILOT

Le langage PILOT est basé sur la notion d'action. Deux types d'actions sont distingués (voir figure1) : les *actions élémentaires*, qui ont leur propre fin et qui se terminent généralement lorsque leur objectif est atteint, et les *actions continues* dont la fin est déclenchée par une autre primitive du langage.

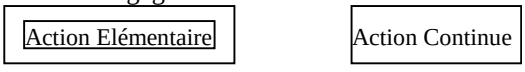


Figure 1. Actions élémentaire et continue

Le langage PILOT fournit différentes structures de contrôle pour la construction de plans :

- La séquentialité : elle est formée d'une suite, éventuellement vide, d'actions et/ou de structures de contrôle. La figure 2 montre un exemple de séquence comportant deux actions élémentaires action1 et action2. L'exécution de l'action2 commence après la fin de l'action1.

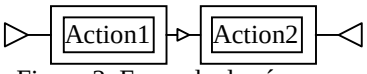


Figure 2. Exemple de séquence

- 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, quand l'une de ses séquences atteint sa fin, elle déclenche l'arrêt des autres séquences et la fin de la préemption.

2.2. Le système de contrôle de PILOT

Le système de contrôle est l'interface entre l'utilisateur et la machine pilotée. Il comporte six modules concurrents : une *interface homme-machine (IHM)*, un *interpréteur*, un *serveur de communication*, un *générateur de règles*, un *évaluateur* et un *module d'exécution* ou *driver*. Ces modules communiquent par socket et par mémoire partagée.

L'*interpréteur* lit le plan stocké en zone de mémoire partagée par l'*IHM* et envoie des ordres aux autres modules afin de réaliser l'exécution du plan. Son fonctionnement peut être résumé comme suit :

Début Interpréteur
    Créer structures pour l'exécution du plan
    Initialiser le lien de communication
    Répéter
        TraitementDeMessage
    Fin Répéter
Fin Interpréteur

TraitementDeMessage est une procédure qui gère l'attente et le traitement des messages reçus par l'interpréteur. Ce dernier est donc un *système dynamique à événement discrets* (SDED) qui évolue au rythme des messages qu'il reçoit : ordre de démarrage de l'exécution du plan, notification de la fin d'exécution d'une action, etc. Cette caractéristique (SDED) s'applique également à l'interprétation des plans. Par exemple, dans l'interprétation de la séquence de la figure 2, c'est la réception de l'événement « fin de la première action » qui va déclencher l'exécution de l'action suivante.

Après cette présentation du langage PILOT et de son système de contrôle, nous présentons brièvement, dans la section suivante, les principaux formalismes de modélisation des SED et justifions le choix des RdP comme support pour la modélisation des algorithmes d'interprétation de plans PILOT.

3. LES RESEAUX DE PETRI COMME FORMALISME DE MODELISATION

Il existe trois types fondamentaux de **modèles de SED** (Zhou *et al.*, 1999) : les modèles de *file d'attente*, les modèles *état-transition* dont les représentants sont les *machines à états* et les *RdP*, et les modèles *orientés objets*. Les modèles de files d'attente offrent des modèles mathématiquement concis permettant de développer des solutions analytiques pour une première prise de décision rapide sous certaines hypothèses restrictives. Il peut toutefois être difficile de représenter certains systèmes à l'aide de files d'attente. Le développement de modèles hiérarchiques avec différents niveaux de détail n'est pas non plus simple avec cette approche. Les modèles état-transition sont plus faciles à faire correspondre aux systèmes en général, et permettent une validation facile d'un modèle de simulation. Les modèles état-transition peuvent être utilisés pour la modélisation hiérarchique afin de faciliter la compréhension du système et d'effectuer une simulation efficace. A l'intérieur des modèles état-transition, les automates finis ont tendance à être très complexes pour des systèmes industriels réalistes. Ils ont également une capacité de modélisation limitée pour la représentation explicite d'opérations concurrentes. Les RdP offrent un outil plus puissant pour gérer la dynamique des événements discrets (Alla *et al.*, 1985) et sont en général plus compacts. Les RdP colorés offrent un modèle plus compact que les RdP ordinaires. Les modèles orientés objet résultent de l'application de la technologie orientée objet à la modélisation et à la simulation des SED. Leurs éléments de base sont des objets dont les modèles et interactions peuvent être représentés à l'aide des autres approches décrites ci-dessus.

Parmi les approches sus-mentionnées, notre choix s'est porté sur les RdP et plus particulièrement les RdP colorés pour différentes raisons :

- Leur nature graphique qui offre la convivialité souhaitée dans le but d'utiliser ultérieurement le modèle comme médium de communication entre les différentes personnes impliquées dans le développement du logiciel de contrôle, afin d'éviter des erreurs telles que la distorsion de l'information mentionnée dans (Nana Tchamnda *et al.*, 2002).

- Ils permettent de représenter relativement simplement les différents concepts de l'algorithmique et de la programmation (séquence, itération, conditionnelle, parallélisme, mécanismes de synchronisation, etc.). Les RdP colorés permettent, contrairement aux RdP ordinaires, de modéliser l'affectation générale, les structures de données ou les conditions booléennes pouvant apparaître dans les synchronisations.
- Ils permettent d'avoir des modèles compacts. La compacité est encore plus importante avec les RdP colorés. Cette caractéristique est très utile pour décrire de façon concise les mécanismes et structures complexes de la programmation. Avec les autres approches, l'on arriverait très vite à des modèles volumineux et difficilement exploitables.
- La disponibilité d'outils pour la simulation et la vérification.
- Leur potentiel pour l'analyse mathématique qui permet de vérifier certaines propriétés du système (atteignabilité d'un état, blocages, etc.).
- Le modèle de RdP est à objectifs multiples. Il peut servir à la fois pour l'évaluation de propriétés comportementales et pour d'autres besoins tels que la génération systématique de code ou encore l'évaluation de performances. Cette caractéristique est une de celles jugées utiles dans un formalisme de modélisation (Ferray-Beaumont et Gentil, 1989).

Différents travaux visant à modéliser des logiciels ou des programmes à l'aide des RdP ont déjà été effectués : génération automatique et vérification de programmes de contrôle (Krogh *et al.*, 1988), conception de logiciels de contrôle robotique (Caloini *et al.*, 1998), analyse de programmes Ada par leur traduction en RdP (Mandrioli *et al.*, 1985), (Helmbold et Luckham, 1985), (Shatz *et al.*, 1989), (Murata *et al.*, 1989), (Tu *et al.*, 1990), (Kaiser et Pradat-Peyre, 1997), (Bruneton et Pradat-Peyre, 1999). Les travaux sur l'analyse de programmes Ada par leur traduction en RdP sont plus en rapport avec la problématique abordée dans cet article. Ils se limitent toutefois à la modélisation des structures relatives à la concurrence telles que les tâches, les objets protégés, les appels d'entrées, etc. Dans notre approche, aucune restriction n'est a priori faite sur l'ensemble des structures de programmation à modéliser.

Après la présentation du langage PILOT, l'illustration du fonctionnement de l'interpréteur et l'argumentation du choix des RdP colorés comme formalisme de modélisation, nous présentons, dans la section qui suit, la modélisation et le test des algorithmes d'interprétation de plans PILOT à l'aide des RdP colorés en environnement Design/CPN.

4. MODELISATION ET TEST DES ALGORITHMES D'INTERPRETATION

Avant de présenter la modélisation et le test des algorithmes d'interprétation proprement dits, il est important, voire indispensable, d'examiner sous quelle forme seront représentés les plans utilisés pour la simulation et le test de ces algorithmes.

4.1 Modélisation du plan

Un plan est formé d'une liste de nœuds. Ces nœuds peuvent représenter une action (élémentaire ou continue), un début de séquence, une fin de séquence, une structure parallèle, une préemption, une conditionnelle, une itération ou encore une structure interne à ces derniers.

Pour définir un nœud, nous avons besoin d'un identificateur, d'un type, des identificateurs des nœuds englobant et suivant. Pour le parallélisme, la préemption, l'itération et la conditionnelle, on a également besoin d'accéder à leurs structures internes (séquences internes, branches de la conditionnelle, nombre d'itérations). Pour répondre à ce besoin une composante appelée argument a été ajoutée à la structure du nœud.

Compte tenu de ces éléments, chaque nœud du plan est représenté par un quintuplet (i_n,t,a,i_e,i_s) où i_n est son identificateur, t son type, a son argument, i_e l'identificateur de son englobant et i_s l'identificateur du nœud qui le suit (l'absence de suivant est matérialisée par l'identificateur –1). Le moins unaire sur les nombres est représenté sous Design/CPN par le symbole tilde (~).

Dans les sous-sections qui suivent, nous présentons les différents champs d'un nœud, en commençant par le champs *type* compte tenu de son utilité pour les autres composantes du nœud.

4.1.1 Le type d'un nœud

Les types de nœuds utilisés dans la modélisation du plan sont les suivants: **Se** (début de séquence), **Sf** (fin de séquence), **Ac** (action continue), **Ae** (action élémentaire), **Pa** (parallélisme), **Pe** (préemption), **Cd** (conditionnelle), **It** (itération), **Lb** (liste de branches d'une conditionnelle), **Bc** (branche d'une conditionnelle, formée d'une condition suivie d'une séquence).

4.1.2. Les identificateurs de nœuds

Ils permettent d'identifier de façon unique un nœud dans le plan. L'identificateur de l'englobant d'un nœud permet d'accéder au nœud représentant la structure qui le contient:

- Le nœud « début de séquence principal » (celui qui commence le plan) n'a pas d'englobant. Il est le seul à ne pas en avoir. L'identificateur utilisé pour indiquer l'absence d'englobant est -1.
- Les autres nœuds « début de séquence » ont comme englobant la préemption, le parallélisme, l'itération ou la conditionnelle qui les contient directement (c'est à dire au premier niveau d'imbrication).
- Les nœuds « Listes de branches » (type Lb) et « Branche conditionnelle » (type Bc) ont pour englobant la conditionnelle qui les contient.
- Les autres nœuds (types Sf, Ac, Ae, Pa, Pe, It, Cd) ont comme englobant la séquence qui les contient

immédiatement. L'identifiant d'englobant adopté dans ce cas est celui du nœud de début de séquence.

4.1.3 L'argument d'un nœud

L'argument est une valeur entière dont la fonction dépend du type de nœud. Pour un nœud de type:

- **Pa** ou **Pe** : l'argument est l'identificateur de la première séquence interne (premier niveau d'imbrication).
- **It** : l'argument représente le nombre de boucles ou la condition d'itération (entier positif ou nul dans le cas d'une itération fixe, -1 pour une itération conditionnelle).
- **Se** : l'argument est l'identificateur du nœud « Branche conditionnelle » pour les séquences d'une conditionnelle (premier niveau d'imbrication) ou l'identificateur du nœud représentant la séquence suivante dans le cas d'un parallélisme ou d'une préemption.
- **Cd** : l'argument est l'identificateur du nœud « liste de branches » pointant sur la première branche de la conditionnelle.
- **Lb** : l'argument est l'identificateur du nœud « Branche conditionnelle » sur lequel pointe le nœud « liste de branches ».
- **Bc** : l'argument représente l'expression de la condition contenue dans la branche conditionnelle. L'argument prend la valeur 0 lorsque l'expression est « faux », 1 lorsque l'expression est « vrai » et 2 lorsque l'expression est autre.
- Autre : l'argument est non utilisé.

4.1.4 Exemples de modèles de plans

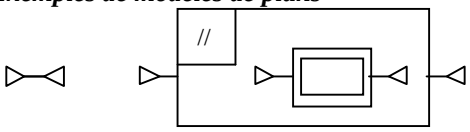


Figure 3.

Figure 4.

Les plans des figures 1 et 2 peuvent être modélisés respectivement par les listes de nœuds [(0,Se,?,-1,1), (1,Sf,?,0,-1)] et [(0,Se,?,-1,1), (1,Pa,2,0,5), (2,Se,-1,1,3), (3,Ae,?,2,4), (4,Sf,?,2,-1), (5,Sf,?,0,-1)].
Le symbole « ? » a été utilisé pour indiquer la non utilisation de l'argument. En pratique, une valeur entière (généralement 0) est utilisée pour respecter la contrainte de type de l'argument. Le nœud début de séquence principal a par convention l'identificateur 0.

4.2. Modélisation des algorithmes d'interprétation

Après la présentation de la modélisation du plan, nous nous intéressons, dans cette partie, à la modélisation des algorithmes d'interprétation. Cette section commence par la présentation de l'approche générale adoptée pour la modélisation. La description des éléments communs à la modélisation des différents algorithmes est ensuite effectuée : modélisation des variables, modélisation des états d'exécution d'un nœud, déclaration des ensembles de couleurs et des variables utilisées, présentation des places fusionnées.

4.2.1 Approche générale de modélisation

Nous avons adopté une approche modulaire afin d'avoir une bonne lisibilité et de faciliter l'utilisation du modèle. Le modèle est formé d'une page de déclarations comportant la définition des couleurs et des variables utilisées tout au long de la modélisation, d'une page comportant le RdP d'initialisation et d'une page par algorithme d'interprétation (séquence, action, itération, conditionnelle, parallélisme, préemption) contenant le RdP correspondant. Les différents RdP communiquent grâce à des places communes, appelées **places fusionnées** dans le jargon de l'outil Design/CPN. Une place fusionnée est une place qui a plusieurs représentations graphiques, sur la même ou différentes pages. Les marques initiales ainsi que les couleurs doivent être les mêmes sur chacune des instances de la place fusionnée. Les noms des instances ne sont pas forcément identiques. Les places fusionnées utilisées dans la modélisation des algorithmes d'interprétation sont au nombre de 5 et constituent les pierres angulaires de la modélisation.

4.2.2 Modélisation des variables

L'approche consiste à créer un jeton coloré pour chaque instance de variable. Le cycle de vie et la portée du jeton (création, lecture, modification de valeur, destruction) reflètent ceux de la variable correspondante. Par exemple, un jeton « pre » crée par l'exécution du modèle d'interprétation de la préemption disparaîtra à la fin de l'exécution de ce dernier.

Dans la modélisation des algorithmes, les variables d'entrée (nœuds du plan, …) sont accédées par des arcs à deux directions (lecture non destructive) contrairement aux variables d'entrée/sortie et aux variables locales pour lesquelles deux arcs distincts sont utilisés dont un sortant de la place qui les contient (lecture) et l'autre entrant vers cette place (écriture).

4.2.3 Les états d'exécution d'un nœud

A chaque nœud du plan en cours d'exécution est associé un **nœud d'exécution** représenté par un jeton comportant à la fois le nœud lui-même, l'état d'exécution du nœud, le mode d'exécution (uniquement dans le cas d'un nœud de type début de séquence), l'identificateur du nœud de préemption englobant et l'identificateur de la liste d'actions continues correspondant à la séquence contenant le nœud. Les états d'exécution possibles sont les suivants :

- ACTIVABLE (ACT): le nœud est prêt à être exécuté. Lorsqu'un nœud devient activable, il est mis dans la place *Traitement* (à l'exception des nœuds fin de séquence auxquels aucun algorithme de traitement n'est associé, et des nœuds début de séquence qui sont placés dans la place *Sequence* en raison de leur traitement particulier dû à la prise en compte du mode d'exécution) afin d'être interprété par le RdP

correspondant à son type (par exemple, le RdP d'interprétation de la préemption pour un nœud dont le champ type vaut **Pe**).

- EN COURS (ENC): le nœud est en cours de traitement.
- DESACTIVE (DES): le traitement du nœud est terminé.

4.2.4 Les ensembles de couleurs et les variables utilisées

Ces couleurs sont définies dans une version du langage ML comportant quelques aménagements spécifiques à l'outil Design/CPN. Les ensembles de couleurs utilisés sont les suivants :

Color I = int ; (* I redéfinit l'ensemble des entiers *)
Color B = bool **with** (faux, vrai) ; (* B redéfinit le type booléen bool *)
Color T = **with** Se | Sf | Ac | Ae | Pa | Pe | Ma | Cd | It | Lb | Bc ; (* type énuméré *)
Color M = **with** KILL | NULL ;
Color E = **with** ACT | ENC | DES ;
Color N = **product** I*T*I*I*I ; (* produit ensembliste *)
Color L = **list** N ; (* type liste d'éléments de type N *)
Color IB = **product** I*B ;
Color IL = **product** I*L ;
Color NEMPC = **product** N*E*M*I*I ;
Color NEPC = **product** N*E*I*I ;

Les déclarations de variables utilisées sont les suivantes :
Var n1, n2 **:** N ; **Var** i1, a1, a2, e1, e2, s1, s2, pre, lc **:** I ; **Var** t1, t2 **:** T ; **Var** l **:** L ;

4.2.5 Les places fusionnées

- La place fusionnée *Plan* : Cette place sert uniquement à stocker les nœuds du plan à interpréter. Son marquage est invariable. En effet, l'interprétation du plan ne modifie pas ce dernier.
- La place fusionnée *Pre* : Cette place stocke toutes les variables de préemption qui sont générées lors de l'interprétation du plan. Chaque variable de préemption *pre* est modélisée par un couple (*i, b*) où *i* est l'identificateur du nœud dont l'interprétation est à l'origine de la création de la variable de préemption et *b* la valeur booléenne indiquant si la condition de préemption est ou non satisfaite (elle est initialisée à faux à la création de la variable *pre*). Les variables *pre* sont créées au début de l'interprétation des nœuds de préemption, à l'exception de celle passée en paramètre lors de l'appel du programme d'interprétation sur le début de séquence principale (c'est-à-dire au début de l'interprétation du plan). Cette dernière a pour identificateur celui de la séquence principale, c'est-à-dire 0.
- La place fusionnée *ActionsC* : On retrouve dans cette place les variables *liste_cont* permettant d'accéder aux actions continues en cours d'exécution dans un parallélisme, une préemption ou une séquence exécutée en mode KILL. Une variable *liste_cont* est représentée par un couple (*i, l*) où *i* est un identificateur et *l* une liste d'actions continues. La variable *liste_cont* associée à une séquence KILL (respectivement à un parallélisme, à une préemption) a comme identificateur celui de la séquence (respectivement celui du parallélisme, de la préemption). La liste qui lui est associée est formée des actions continues en cours d'exécution dont elle (respectivement l'une des séquences internes au premier niveau d'imbrication du parallélisme, de la préemption) est l'englobant immédiat.
- La place fusionnée *Sequence* : Dans cette place, on ne trouve que les nœuds d'exécution correspondant aux débuts de séquences. Ces nœuds d'exécution se distinguent des autres par la présence de la composante mode d'exécution qui permet de différencier les séquences exécutées en mode KILL des séquences exécutées en mode NULL. Une séquence exécutée en mode KILL termine ses actions continues contrairement à une séquence exécutée en mode NULL.
- La place fusionnée *Traitement* : La place *Traitement* modélise l'exécution de la fonction traitement. Dans cette place, on traite les nœuds qui se trouvent dans une séquence en cours de traitement. A chacun de ces nœuds est associé un nœud d'exécution dans la place *Traitement*. Les nœuds d'exécution de cette place ont la même constitution que ceux de la place *Sequence*, à l'exception du champs « mode d'exécution » qu'ils ne comportent pas.

Après cette présentation des éléments communs aux différentes modélisations, nous présentons ci-dessous la modélisation de l'algorithme d'interprétation de la séquence. Pour des raisons de concision, les RdP des autres algorithmes d'interprétation ne sont pas présentés (actions élémentaire et continue, conditionnelle, itération, parallélisme et préemption).

4.3. Modélisation de l'algorithme d'interprétation de la séquence

4.3.1 L'algorithme d'interprétation de la séquence KILL (resp. NULL)

Paramètres locaux (resp. **Entrées/Sorties**)
    liste_cont : liste des actions continues
Entrées
    pre : booléen
Paramètres locaux
    prim_courante : pointeur sur la primitive courante
Début
    Tq prim_courante ≠ *nul et non* (pre)
        Si prim_courante→type = 4 /*action continue*/
            Alors ajouter(prim_courante, liste_cont)
        Fin_si
        traitement(prim_courante, pre)
        prim_courante ← prim_courante→suivant
    Fin_tq
terminer(liste_cont) /*Terminaison des actions continues*/ (resp. rien)
Fin

La figure 5 montre le RdP modélisant l'algorithme d'interprétation de la séquence. Nous le décrivons ci-après.

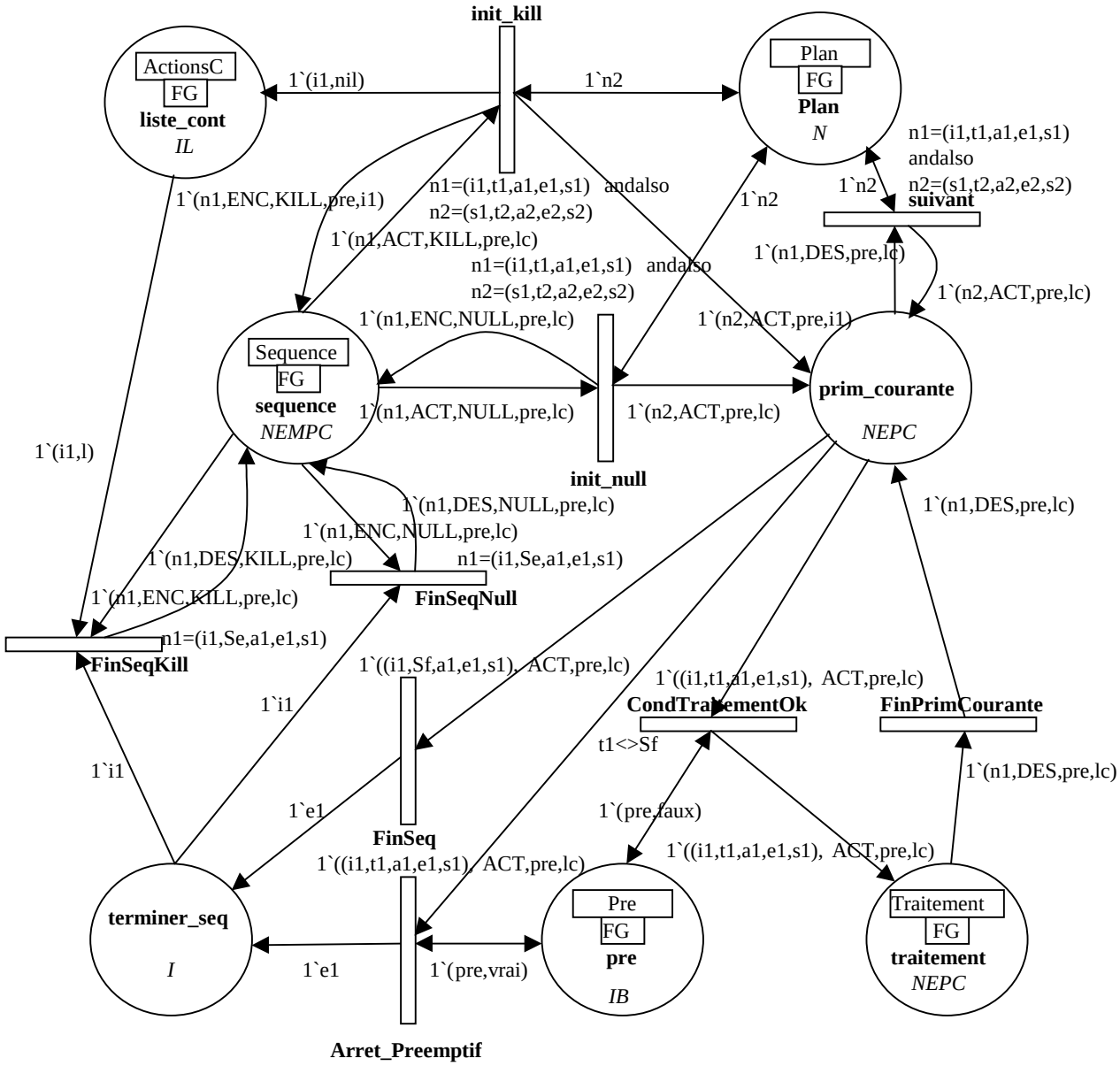


Figure 5. Réseau de Petri modélisant l'algorithme d'interprétation de la séquence

4.3.2 Description du RdP modélisant l'algorithme d'interprétation de la séquence

Les places

Comme on peut le noter, le modèle proposé comporte, en plus des 5 places fusionnées présentées plus haut, 2 places spécifiques *prim_courante* et *terminer_seq*. La place *terminer_seq* sert à stocker les identificateurs des séquences à stopper. La place *prim_courante* modélise quant à elle la variable « prim_courante ». Sa couleur est la même que celle de la place Traitement.

Les variables

Chaque variable « pre » (respectivement « liste_cont », « prim_courante ») est représentée par un jeton stocké dans la place *Pre* (respectivement *ActionsC*, *prim_courante*).

Les Initialisations

Une séquence est exécutée lorsqu'un nœud d'exécution de type « début de séquence » se trouve dans la place *sequence* dans l'état ACT (activable). Avant de lancer l'algorithme proprement dit, les variables sont initialisées en fonction du mode d'exécution de la séquence :

- mode NULL (transition init_null): le noeud suivant est mis dans la place *prim_courante*.
- mode KILL (transition init_kill): la variable « liste_cont » est créée et le nœud suivant est mis dans la place *prim_courante*.

Dans les 2 cas, l'état d'exécution de la séquence passe à ENC (en cours).

Gestion du test de la boucle « Tant que » (Tq)

La gestion de la boucle **Tq** intervient dès que la primitive courante est activable dans la place *prim_courante*. A ce niveau, 3 cas peuvent se présenter :

1. La séquence est préemptée c'est-à-dire « pre » a la valeur vrai (transition Arret_Preemptif): l'identifiant de la séquence est mis dans la place *Terminer_seq*.
2. La primitive courante est une « fin de séquence » (transition Fin_Seq) : l'identifiant de la séquence est mis dans la place *Terminer_seq*. Ce cas correspond au test de la condition *prim_courante ≠ nul*.
3. La séquence n'est pas préemptée et la primitive courante n'est pas une « fin de séquence » (transition Cond_Traitement_Ok) : le noeud est mis dans la place *Traitement*. On peut noter la transmission du paramètre « pre »; celle de « lc » est nécessaire, bien que ce dernier ne soit pas présent dans l'appel *traitement (prim_courante, pre)*. En effet, étant donné que la place « traitement » est une place fusionnée utilisée pour différents appels de la fonction traitement dont certains requièrent le paramètre liste_cont, il convient d'avoir une même interface d'appels afin de répondre à la contrainte sur le type de couleur associé aux différentes instances d'une place fusionnée (le type de couleur doit être le même).

Gestion du Traitement

La gestion du **Si** est faite au niveau du traitement. Suivant le type du nœud, le RdP correspondant l'exécute. A la fin de l'exécution d'un nœud, ce dernier devient désactivé.

Passage au nœud suivant

Le passage au nœud suivant intervient lorsque la primitive courante est désactivée dans la place *prim_courante* (transition « suivant ») : le nœud suivant est alors mis dans la place *prim_courante* dans l'état activable (ACT).

Gestion de la terminaison d'une séquence

Quand un identificateur est dans la place Terminer_seq, la séquence correspondante passe à l'état désactivé (transitions Fin_Seq_Null et Fin_Seq_Kill). Par ailleurs, dans le cas d'une séquence exécutée en mode KILL (transition Fin_Seq_Kill), les actions continues correspondantes sont supprimées de la place ActionsC (*liste_cont*).

4.4. Les tests

Afin de tester les algorithmes d'interprétation, des plans de tests ont été générés en s'appuyant sur l'approche décrite dans (Nana Tchamnda *et al.*, 2002). Les initialisations suivantes sont effectuées avant la simulation :

- La place *Plan* est initialisée avec le modèle de plan dont on veut simuler l'interprétation (plan de test).
- La place *Pre* est initialisée avec un jeton de préemption 1`(0,faux). Ce jeton est utilisé comme paramètre d'entrée pour l'interprétation de la séquence principale.
- La place *Sequence* est initialisée avec un jeton représentant le nœud d'exécution associé à la séquence principale dont le mode d'exécution est positionné à KILL et le champ état à ACT.

Un échantillon des plans de test appliqués au modèle d'interprétation est donné dans le tableau 1. Seuls les plans jugés utiles pour la bonne compréhension de l'analyse des résultats de simulation y figurent.

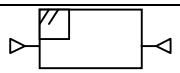
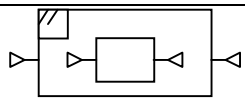
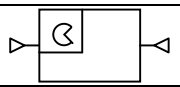
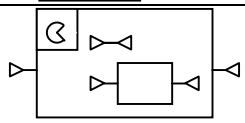
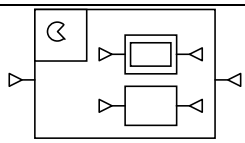
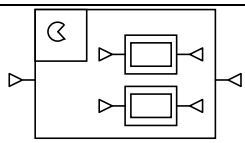
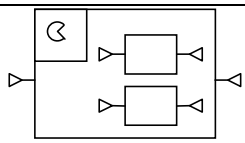
No	Plan	Modèle Design/CPN
8		1`(0,Se,0,~1,1)+1`(1,Pa, ~1,0,2)+1`(2,Sf,0,0,~1)
11		1`(0,Se,0,~1,1)+1`(1,Pa, 2,0,5)+1`(2,Se,~1,1,3)+ 1`(3,Ac,0,2,4)+1`(4,Sf, 0,2,~1)+1`(5,Sf,0,0,~1)
20		1`(0,Se,0,~1,1)+1`(1,Pe, ~1,0,2)+ 1`(2,Sf,0,0,~1)
28		1`(0,Se,0,~1,1)+1`(1,Pe,2, 0,7)+1`(2,Se,4,1,3)+1`(3, Sf,0,2,~1)+1`(4,Se,~1,1,5) +1`(5,Ac,0,4,6)+1`(6,Sf,0, 4,~1) + 1`(7,Sf,0,0,~1)
29		1`(0,Se,0,~1,1)+1`(1,Pe,2, 0,8)+1`(2,Se,5,1,3)+1`(3, Ae,0,2,4)+1`(4,Sf,0,2,~1)+ 1`(5,Se,~1,1,6)+1`(6,Ac,0, 5,7)+1`(7,Sf,0,5,~1)+1`(8, Sf,0,0,~1)
30		1`(0,Se,0,~1,1)+1`(1,Pe,2, 0,8)+1`(2,Se,5,1,3)+1`(3, Ae,0,2,4)+1`(4,Sf,0,2,~1)+ 1`(5,Se,~1,1,6)+1`(6,Ae,0, 5,7)+1`(7,Sf,0,5,~1)+1`(8, Sf,0,0,~1)
31		1`(0,Se,0,~1,1)+1`(1,Pe,2, 0,8)+1`(2,Se,5,1,3)+1`(3, Ac,0,2,4)+1`(4,Sf,0,2,~1)+ 1`(5,Se,~1,1,6)+1`(6,Ac,0, 5,7)+1`(7,Sf,0,5,~1)+1`(8, Sf,0,0,~1)

Tableau 1. Echantillon des plans de tests

La troisième colonne du tableau donne le marquage initial de la place fusionnée *Plan*. Il s'agit de la représentation, sous forme de multi-ensemble (forme des marquages sous Design/CPN), du modèle de plan construit conformément à l'approche décrite en section 2.

5. RESULTATS DE LA SIMULATION ET ANALYSE

La simulation de l'exécution du plan de test 11 conduit à s'interroger sur le sens d'un tel plan. En effet, bien que son exécution se termine bien, l'action continue est arrêtée par la fin de la séquence contenant l'action continue elle-même, c'est-à-dire juste après le lancement de l'exécution de ladite action. La simulation de l'exécution du plan de test 20 conduit quant à elle à un blocage dans l'attente de la fin d'une première séquence inexistante dans ce plan ; l'on s'attendrait pourtant à une terminaison immédiate comme dans le cas d'un parallélisme sans séquence interne (cas du test 8). Le test 28 révèle également une anomalie dans l'algorithme de la préemption : la séquence contenant l'action continue peut déclencher l'arrêt de l'autre séquence de la

préemption. Le même problème subsiste dans les autres tests relatifs à la préemption (test 29, 30 et 31). Les autres tests (ceux ne figurant pas dans le tableau 1) n'ont révélé aucune anomalie lors de la simulation.

6. CONCLUSIONS ET PERSPECTIVES

La modélisation nous a permis de constater que des simplifications sont envisageables tant au niveau de la représentation interne du plan qu'au niveau des algorithmes d'interprétation. En ce qui concerne la structure interne du plan, l'une des simplifications envisageables est la suppression du noeud « liste_seq ». En ce qui concerne les algorithmes d'interprétation, la suppression des modes d'exécution de séquence est envisageable et permet en particulier de fusionner les algorithmes de traitement de séquence KILL et NULL. Cette simplification a été effectuée dans la version actuelle de l'interpréteur. Les plans tels que celui du test 11 font partie des plans désormais rejetés par la syntaxe du langage et par le mécanisme de construction et de vérification incrémentales de plans. De même, dans la syntaxe révisée du langage, une préemption contient, comme tout parallélisme, au moins une séquence dite normale (sans action continue au premier niveau), ce qui écarte le problème observé au niveau de la simulation de l'exécution du test 20. Dans (Nana Tchamnda *et al.*, 2002), nous avons mentionné que la mauvaise connaissance d'un langage est une source potentielle d'erreurs logicielles. Les modèles d'algorithmes d'interprétation réalisés au cours de ce travail pourront être utilisés, après validation, comme support d'apprentissage de la sémantique du langage et du fonctionnement interne des algorithmes d'interprétation, pour les personnes amenées à modifier le code source de l'interpréteur. Nous étudions actuellement la conformité des algorithmes d'interprétation par rapport à la sémantique opérationnelle du langage, la suite envisagée étant la régénération du code à partir des modèles d'interprétation validés.

REFERENCES

Alla, H., P. Ladet, J. Martinez, and M. Silva, 1985. *Modeling and validation of complex systems by colored Petri nets : application to a flexible manufacturing system*, Advances in Petri Nets 1984, G. Rozenberg, H. Genrich, and G. Roucairol (Eds.), Springer-Verlag, pp. 15-31.

Bruneton, E. and J.-F. Pradat-Peyre, 1999. *Automatic Verification of Concurrent Ada Programs*, Ada-Europe'99 International Conference on Reliable Software Technologies, Santander, Spain, pp. 146-157.

Caloini, A., G. Magnani and M. Pezze, 1998. *A technique for designing robotic control systems based on Petri nets*, IEEE Trans. on Control Systems Technology, 6(1), pp. 72-87

Ferray-Beaumont, S. and S. Gentil, 1989. *Declarative Modelling for process supervision*, Revue d'intelligence artificielle, vol 3, no 4.

Fleureau, J.L., 1998. *Vers une méthodologie d'un système de programmation de télérobotique: comparaison des approches PILOT et GRAFCET*, PhD thesis, Université de Rennes 1, France.

Helmbold, D. and D. Luckham, 1985. *Debugging Ada-tasking programs*, IEEE Transactions on Software Engineering, Vol. 2 (No. 2) : 45-57.

Kaiser, C. and J.F. Pradat-Peyre, 1997. *Comparing the reliablity provided by tasks or protected objects for implementing a resource allocation service : a case study*. In Tri'Ada, St Louis, Missouri. ACM SIGAda.

Krogh, B. H., R. Willson and D. Pathak, 1988. *Automated generation and evaluation of control programs for discrete manufacturing processes*. Proc. of 1988 Int. Conf. On Computer Integrated Manufacturing, Troy, NY, pp. 92-99.

Le Rest, E., 1996. *PILOT: un langage pour la télérobotique*, PhD thesis, Université de Rennes 1, France.

Le Rest, E., J.L. Fleureau and L. Marcé, 1997. *PILOT: semantics and implementation of a language for telerobotics*, In IROS'97, IEEE, Grenoble, France.

Mandrioli, D., R. Zicari, C. Ghezzi, and F. Tisato, 1985. *Modeling the Ada task system by Petri nets*, Computer Languages, Vol. 10 (No. 1) : 43-61.

Murata, T., B. Shenker, and S. M. Shatz, 1989. *Detection of Ada static deadlocks using Petri nets invariants*, IEEE Transactions on Software Engineering, vol.15 (No. 3) : 314-326.

Nana Tchamnda, L. and L. Marcé, 2001. *Vers une programmation sûre avec PILOT*, MSR'2001, Colloque Francophone sur la Modélisation des Systèmes Réactifs, Toulouse, France.

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

Shatz, S. M., K. Mai, D. Moorthi, and J. Woodward, 1989. *A toolkit for automated support of Ada-tasking analysis*, In Proceedings of the 9th Int. Conf. On Distributed Computing Systems, pp. 595-602.

Tu, S., S. M. Shatz, and T. Murata, 1990. *Applying Petri nets reduction to support Ada-Tasking deadlock detection*, In Proceedings of the 10th IEEE Int. Conf. On Distributed Computing Systems, pp. 96-102, Paris, France.

Zhou, M. C. and K. Venkatesh, 1999. *Modeling, simulation and control of flexible manufacturing systems, A Petri Net Approach*. Series in intelligent control and intelligent automation, Vol. 6., Word Scientific Publishing.