YALTA : un langage pour la teleoperation

Jean-Christophe Paoletti, Lionel Marcé

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1258

Programme 6
Robotique, Image et Vision

YALTA : UN LANGAGE POUR LA TELEOPERATION "

Jean-Christophe PAOLETTI
Lionel MARCÉ

Juin 1990

RR-1258

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

YALTA : un langage pour la téléopération"

Jean-Christophe PAOLETTI
IRISA / INRIA - Rennes
Campus de Beaulieu
F-35042 RENNES Cedex FRANCE
Tel. (33)-99.36.20.00
e-mail paoletti@irisa.fr

Lionel MARCÉ
LIB / UBO - Brest
6 avenue Victor Le Gorgeu
F-29283 Brest Cedex FRANCE
Tel. (33)-98.31.61.21
e-mail marce@irisa.fr

Résumé

Dans le cadre des travaux sur la télérobotique et, plus généralement des applications nécessitant une forte interactivité avec l'opérateur humain, nous proposons un langage graphique (YALTA : Yet Another Language for Telerobotics Applications) pour l'aide à l'opérateur. Ce langage lui permettra de décrire un plan, éventuellement de le simuler, puis de contrôler son exécution.

Grâce à son caractère visuel et au nombre limité de structures de contrôle (conditionnel, parallélisme, alternative, iteration, réactivité) YALTA est très simple d'utilisation. Sa sémantique opérationnelle sera exposée. Elle a été générée à partir de la notion de système de processus synchronisés.

Le plan est considéré comme un système réactif. Il réagit aux signaux d'entrées et aux alarmes prévenant de son environnement en envoyant des signaux de sorties pour déclencher une action du robot ou l'activation d'une règle. L'opérateur peut intervenir directement, pendant l'exécution, sur la structure graphique de YALTA pour ajuster les paramètres d'une action, ou pour modifier dynamiquement une partie du plan.

Un système de règles est associé au plan et aux actions, pour vérifier, durant l'exécution, l'intégrité de certaines contraintes.

YALTA : Yet Another Language for Telerobotics Applications

Abstract

In the frame work of telerobotics and, more generally of applications which need a strong human interactivity, we propose a visual language (YALTA : Yet Another Language for Telerobotics Applications) as an aid to the operator. This language allows him to describe a plan, possibly to simulate it, then to monitor its execution.

Thanks to its visual features and a limited number of control structures (conditional, parallel, alternate, iteration) YALTA is very simple to use. Its operational semantics will be explained. It has been produced from the concept of synchronized processes system.

The plan is considered like a reactive system. It reacts to input signals and alarms coming from its environment by answering output signals to trigger a robot action or a rule activation. The operator can act directly, during the execution, on the visual structure of YALTA to adjust parameters of an action, to modify dynamically a part of the plan.

A system of rules is associated with the plan and the actions, to verify, during the execution, some integrity constraints.

Table des matières

1 Introduction

Dans un certain nombre de cas, tels que les ateliers de production l'ensemble des différentes configurations du système est restreint. L'ensemble des accidents et des configurations de reprise est de taille finie et relativement réduite. Une programmation dans l'optique d'une exécution en mode automatique peut être envisagée. Des outils, tels que Grafcet, réseaux de Pétri peuvent alors être employés pour décrire et pour contrôler le fonctionnement du système [1, 2].

Mais dans de nombreux autres cas, ceci est irréalisable : robotique mobile [3], robot d'intervention [4] (incendie, accident nucléaire ...), bras sur une navette spatiale [5] ... L'environnement évoluant au cours du temps, de manière souvent imprévisible, le mode téléopéré est choisi. L'opérateur reste maître de la machine et la dirige dans ses moindres mouvements. Il agit en fonction des évolutions de l'environnement avec son savoir faire.

L'étude de l'automatisation dans le cadre de la robotique mobile est en cours [7], mais pour le moment l'environnement du robot reste encore très structuré (escalier, couloir, sol plan). Cette étude se limite souvent aux déplacements du robot : calcul de trajectoires, évitement d'obstacles. La planification de tâches (c'est-à-dire la contruction automatique d'un plan d'exécution à partir du but à atteindre) reste plus que problématique, du fait des évolutions imprévisibles de l'environnement et de l'explosion du nombre des états du système : robot, environnement.

Entre la robotique automatique et la téléopération, il existe des applications qui appartiennent au domaine de la téléopération assistée par ordinateur [8] (T.A.O.). Pour ce type d'applications, nous considérons que l'homme est constamment présent lors d'une session, comme :

- concepteur : création du plan, construction des modèles relatifs au robot et à l'environnent.
- superviseur : exécution en mode semi-automatique et contrôle de l'exécution.
- opérateur : exécution en mode téléopéré.

Nous proposons, pour ce type d'applications, un langage graphique d'aide au contrôle d'exécution : YALTA [16]. Ce langage fait suite au constat du manque d'outils informatiques de haut niveau pour gérer les interactions entre un robot, un opérateur et un environnement [9]. Il va permettre :

- de programmer un plan d'actions ,de façon simple et la moins experte possible.
- de simuler l'exécution du plan.
- d'effectuer un contrôle automatique de l'exécution.
- de le modifier dynamiquement.
- de reprendre un mode téléopéré en cas d'accidents ou lors d'actions délicates.
- d'effectuer une visualisation de l'exécution.

Nous allons présenter l'organisation et le fonctionnement d'un système de contrôle d'exécution tel que nous le concevons (section 2). Puis nous exposerons le modèle d'exécution (système de processus synchronisés) du langage YALTA (section 3), ainsi que sa syntaxe et sa sémantique. Avant de conclure (section 5), nous exposerons deux exemples de plans construits à l'aide de YALTA (section 4).

2 Organisation et Fonctionnement du Système de Contrôle d'Exécution

Un système de téléopération assistée par ordinateur [8] est divisé en trois parties :

- un univers maître : organes de commande et de restitution.
- un univers virtuel : électronique et informatique de commande.
- un univers esclave : organes opératifs et sensoriels, outils et objets de l'environnement.

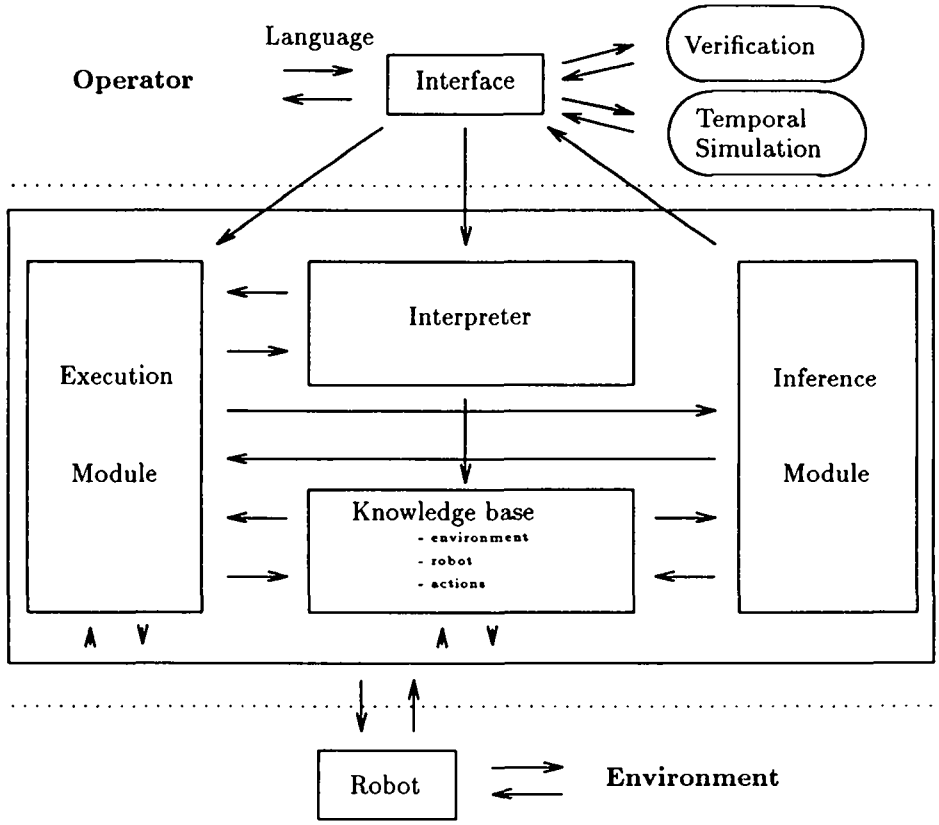

Figure 1 : Organisation du système

Le rôle d'un tel système est de donner la possibilité à un opérateur humain d'intervenir à distance en milieu hostile dans le cadre de tâches variées et non répétitives.

Le contrôleur d'exécution *(Figure 1)* se situe entre l'opérateur (l'univers maître) et le robot (l'univers esclave). Lors d'une session normale, l'opérateur, à l'aide d'une interface graphique, communique son plan au contrôleur d'exécution. Ce plan peut être préalablement simulé, pour vérifier qu'il peut en exister au moins une bonne exécution. On peut aussi effectuer une vérification sur la cohérence lors de modifications effectuées en ligne. L'ensemble : *langage, interface, simulateur et module de vérification* constitue l'environnement utilisateur.

Le plan transmis au contrôleur est constitué de deux parties :

- Un ensemble d'objets décrivant les actions que doit effectuer le plan, chaque objet est une instance d'une classe d'action (par exemple : l'action "aller en A" est une instance de la classe d'action "aller" avec pour valeur de l'attribut "destination" la position A).
- Un programme écrit dans le langage YALTA, ce programme spécifie le déroulement des actions à l'aide d'opérateurs tels que le séquentiel, le parallèle, l'alternative ...

Le plan est alors dirigé vers l'interpréteur qui traduit le programme en un automate pour le module d'exécution et dirige l'ensemble des actions vers la base de connaissances. Pour chaque action, le module d'exécution envoie l'ordre de début d'action au robot et prend en compte, en retour, les messages de terminaison ou les alarmes. *Le module d'exécution, l'interpréteur, l'automate et les actions* correspondent à la partie exécution du contrôle d'exécution.

L'autre partie spécifique au contrôle est constituée de :

- Un *moteur d'inférence* qui doit servir à vérifier l'intégrité de certaines contraintes durant toute ou une partie de l'exécution.

- Une *base de connaissances* qui mémorise l'état des différentes parties du robot et de son environnement, ainsi que les informations (état, contraintes, règles) associées aux actions.

2.1 L'Environnement Utilisateur

2.1.1 Le Langage

YALTA est un langage graphique. Ce trait du langage est induit par les propositions suivantes :

- L'opérateur doit déterminer le résultat global d'un programme rapidement, sans avoir à rentrer dans les détails de ce dernier.
- L'opérateur doit facilement discerner les différentes branches parallèles.
- L'opérateur doit pouvoir modifier dynamiquement certaines parties du programme (ajout ou suppression d'une action, d'une branche parallèle ...).
- La visualisation de l'exécution du programme doit être simple et significative.

Pour éviter de nuire à la compréhension des plans, l'ensemble des constructeurs est réduit. Cela limitera aussi le nombre des représentations possibles pour un même plan et facilitera la communication entre les différents opérateurs. Il est, en effet, indispensable que l'opérateur comprenne la totalité du plan, il en fait partie intégrante que ce soit comme superviseur ou comme actionneur.

YALTA servira à l'opérateur pour créer son plan (un ordonnancement complexe d'actions) et pour agir lors de son exécution : interruption, modification en ligne, intervention lors de pannes et génération d'évènements. Une description complète, tant pour la représentation que pour la sémantique du langage, sera faite ultérieurement.

2.1.2 L'Interface

Une interface graphique est nécessaire à l'édition du langage. Cette interface ne doit pas remettre en cause la simplicité du langage. Mais au contraire, elle doit fournir une aide efficace pour la conception de plans. Plusieurs modules la constituent : l'éditeur de plan, la visualisation de l'exécution, la visualisation de la simulation et la visualisation des différentes parties de la base de connaissances (robots, environnement).

Editeur de plan Ce module permet d'éditer ou de modifier un plan (ou une action composée) à l'aide du langage précédemment introduit. Toutes les fonctions classiques d'un éditeur sont présentes : couper, copier, coller, sauvegarder dans un fichier, lire dans un fichier ... Le module sert aussi à la saisie des règles. L'opérateur aura la possibilité d'ajouter des actions atomiques (avec préconditions, règles et effets) au fur et à mesure que l'on modifie l'application robotique.

Visualisation de l'exécution Ce module visualise les différentes actions actives lors de l'exécution, ainsi que les règles utilisées. Il visualise, si la demande en est faite, tous les niveaux de l'exécution : lors de l'activation d'une action composée, une fenêtre spécifique à l'exécution de cette action s'affiche à l'écran. La visualisation est effectuée sur la structure graphique du plan. Lors d'un incident, l'opérateur saura exactement où et pourquoi il a eu lieu. Le module permettra aussi à l'opérateur de générer des évènements vers le contrôleur d'exécution. Une interruption générale du plan peut être effectuée, ainsi que la reprise d'un plan interrompu. Lors d'une modification en ligne du plan, il sera fait appel à l'éditeur.

Visualisation de la simulation Après sa conception, le plan pourra être simulé temporellement. Ce module visualise à l'aide de graphes temporels les résultats de cette simulation. Pour chaque prédicat vérifiable sur le monde environnant et sur l'état du robot, l'évolution au cours du temps de sa fonction de véracité est donnée graphiquement.

Visualisation de la base de connaissances Ce module sert à visualiser l'état des différentes parties de l'environnement et du robot. Il donne un retour symbolique de l'état de la base de connaissances. Il est utilisé comme outil de modifications (création d'une nouvelle classe d'objet, ajout de ressources dans un stock ...).

2.1.3 La Simulation Temporelle

Une simulation temporelle du plan [10] peut être exécutée après sa conception. Le succès de cette simulation met en évidence le bon déroulement du plan sur l'échelle du temps réel.

La simulation proposée est produite à partir d'une logique temporelle, à base d'intervalles. Un intervalle est un couple d'instants (un instant de début et un instant de fin). Les relations sur l'ensemble des intervalles sont de types *avant*, *égal*, *pendant* ... Les actions sont des structures possèdant un nom, une durée, une liste de préconditions, une liste d'effets négatifs (liste des prédicats, modélisant le monde, rendus faux par l'exécution de l'action), une liste d'effets positifs (liste des prédicats, modélisant le monde, validés par l'exécution de l'action). Le simulateur déroule alors le plan avec la possibilité de faire intervenir des évènements extérieurs.

2.1.4 Vérification

Lors d'une modification du plan en ligne, une vérification est effectuée. Elle permet de savoir quelle partie du plan a été modifiée et si cette modification est valide par rapport à l'état de l'exécution.

Après avoir été conçu par l'opérateur, le plan est interprété sous forme d'automates. L'interprétation rend le plan modulaire : si la modification du plan ne porte que sur une de ses branches parallèles, seul l'automate associé à cette branche sera modifié. La vérification sert alors à savoir quelle partie de l'automate doit être recalculée. De plus, elle interdit toute modification du plan liée à une partie déjà exécutée ou en cours d'exécution.

2.2 Le Contrôleur d'Exécution : Partie Exécution

2.2.1 Les Actions

Les actions sont les objets qui vont permettre à l'opérateur de spécifier les actions exécutées par les effecteurs du robot. Chaque action est une instance d'une classe d'actions et donc hérite des différents attributs de sa classe (état, paramètres, règles ...). Les actions appartiennent à l'un des trois types suivants:

atomiques : Elles se situent au niveau des commandes comprises par le robot. Elles ne sont pas décomposables.

composées : Ce sont des plans construits avec les constructeurs du langage et d'autres actions atomiques et composées.

continues : Ce sont des actions atomiques dont la date de fin est déterminée soit par le déclenchement d'une règle, soit par la terminaison de la structure de contrôle parallèle l'englobant.

A chacune des actions (et donc des plans) peuvent être associées les objets suivants :

Un indicateur d'état Cet attribut indique si l'action est :

- en cours d'exécution.
- en attente.
- inactive.
- interrompue : une modélisation de l'état d'avancement de l'action lors de l'interruption est alors disponible.
- exécutée.

Un intervalle temporel Cet intervalle représente les durées possibles pour la bonne exécution de l'action. Cet intervalle indiquera si l'action est :

- instantanée.
- à durée connue : il sera possible de savoir en fonction de la durée réelle de son exécution si l'action s'est bien déroulée ou non.
- à durée variable : la durée sera déterminée par l'exécution d'une autre partie du plan.

Des paramètres Ils peuvent préciser :

- les modules exécutant l'action.
- le mode d'exécution : manuel, semi automatique, automatique.
- les coordonnées à atteindre.
- des valeurs seuils ...

Un plan d'action Cet attribut n'est valable que pour les classes d'actions composées. Il représente le plan associé à l'action.

Des préconditions Ce sont des conditions qui doivent être vérifiées avant le lancement de l'action. La non validité d'une précondition suffit à retarder le début de l'exécution et à provoquer l'envoi d'une alarme.

Des règles Elles sont à vérifier tout au long de l'exécution de l'action. En cas de non respect, une alarme ou un plan de secours est déclenché. Ces conditions correspondent à des règles locales de l'exécution de l'action, elles ne seront déclenchables que lors de l'exécution de cette classe d'action.

Des effets Ils représentent les effets produits sur l'environnement par l'exécution de l'action. Ces effets sont reportés dans la base de connaissances.

Les préconditions, les règles et les effets liés à une action sont déterminés par l'opérateur ou par un superviseur.

2.2.2 L'Interpréteur

Une fois son plan conçu et simulé, l'opérateur va vouloir en faire une exécution. Pour cela, il lui suffira de transmettre son plan au contrôleur d'exécution. L'interpréteur est alors le module qui va rendre le plan exécutable.

Il est chargé dans un premier temps de transmettre, à la base de connaissances, l'ensemble des instances d'actions associées au plan. Puis, il produit une interprétation du plan sous forme d'automate. Cette étape se fait à l'aide des règles de transition [11] qui définissent la sémantique opérationnelle de YALTA.

L'exécution d'une action sera traduite de la manière suivante : vérification des préconditions, envoi de l'ordre de début d'action, attente du signal de fin d'action ou d'un évènement anormal et enfin traitement des effets de l'action.

L'automate ainsi créé rend le plan modulaire; lors de la modification d'une branche parallèle seul l'automate associé à cette branche sera recalculé.

Le plan sera alors traité comme un module réactif. Un système réactif produit des données traitées en réponse à une succession d'évènements [12, 13]. Ce qui correspond bien au fonctionnement de l'automate qui va réagir aux interruptions (opérateur, système) en produisant des ordres de début d'action ou des alarmes.

2.2.3 Le Module d'Exécution

Le module d'exécution est, en fait, essentiellement composé d'un interfaçage entre les évènements extérieurs et les signaux produits par l'exécution du plan.

Lors du déclenchement d'une action, le plan émet un message du type de *debut-action*. A ce message est associé le nom de l'action. Ceci permet de récupérer dans la base de connaissances les valeurs associées aux différents paramètres de l'action et de les appliquer au corps de celle-ci. Le module produit alors des signaux interprétables par le robot. En retour, lorsqu'il termine l'exécution et qu'il émet un évènement de fin d'action, le module interprète cet évènement et transmet vers le plan un signal du type *fin-action* auquel est associé le nom de l'action concernée. Le module veille constamment aux possibles interruptions provenant du moteur d'inférence ou de l'opérateur. Il permet aussi de fournir à l'interface les données nécessaires à la visualisation de l'exécution.

2.3 Le Contrôleur d'Exécution : Partie Contrôle

2.3.1 La Base de Connaissances

La base de connaissances est un système répondant aux messages : de création et de destruction d'instances ou de classes d'objets, de lecture et de maintenance de variables, de demande d'acquisition et de retour d'acquisition de valeurs. Elle va servir de base au retour symbolique modélisant l'état du monde et du robot, de mémorisation des descriptions des actions à réaliser par le plan, ainsi qu'au fonctionnement du moteur d'inférence.

La base de connaissances comprend une partie représentation du robot et de son environnement. L'organisation de cette partie de la base est hiérarchique et orientée objet. L'attribut **contient** introduit une hiérarchie à l'aide de la relation contenir, cette hiérarchie pourrait être étendue par des relations du type : en dessous, au dessus, à droite ...

La valeur courante d'un paramètre n'est pas forcément contenue dans la base, surtout si elle est soumise à de nombreuses modifications. Le paramètre peut très bien être représenté uniquement par le chemin d'accès à sa valeur (ex: le nom d'un capteur) ou par la procédure pour la calculer.

La deuxième partie de la base contient les classes et les objets associés aux actions du plan. Ces objets sont représentés de la manière vue précédemment avec les attributs : indicateur d'état, intervalle temporel, paramètres, plan d'action, préconditions, règles, effets.

2.3.2 Module d'Inférence

Le module est composé d'un moteur d'inférence qui raisonne sur l'ensemble des règles associées aux actions actives de la base de connaissances. Les règles sont de la forme **si** *condition* **alors** *action*. Elles sont stockées dans les champs, *règles* et *préconditions* des instances des classes d'actions.

Dès que la condition d'une règle est validée, l'action lui correspondant est exécutée par l'envoi d'une alarme vers :

le module d'exécution : si l'action correspond à un plan (écart d'un bras ...).

la base de connaissances : si l'action correspond au changement d'état d'une variable.

l'interface : s'il s'agit d'un message à l'opérateur (manque d'énergie ...).

Le module d'inférence sert aussi à vérifier les préconditions des actions. Avant l'exécution d'une action, une demande de vérification de préconditions est envoyée au module d'inférence. Il y répondra par un message *ok* ou *nok* suivant que la vérification des préconditions sur la base de connaissances a été un succès ou un échec.

Ce système a été décrit de manière à mettre en avant l'aide apportée à l'opérateur. Nous avons commencé par décrire et implémenter le langage graphique autour duquel il s'articule : YALTA.

3 YALTA : Représentation et Sémantique

YALTA est un langage graphique pour l'aide à la programmation et au contrôle d'exécution de plans d'actions en téléopération. Ce langage est donc constitué de structures de contrôle liées à l'informatique (séquence, parallélisme, conditionnelle, itération) et d'autres à la téléopération (actions, alternative, réactivité). Après avoir présenté les notions nécessaires à la compréhension du modèle d'exécution, nous décrirons successivement la représentation graphique et la sémantique des structures de contrôle informatiques, puis celles des structures de contrôle liées à la téléopération. La sémantique opérationnelle de YALTA a été produite en utilisant la méthode développée par Plotkin [11].

3.1 Le Modèle d'Exécution

Le modèle d'exécution, pour un plan réalisé à l'aide de YALTA, dérive de la notion de système de processus synchronisés [14].

3.1.1 Le Modèle d'Arnold et Nivat

Un système de transition sur un alphabet A est un couple $< Q, T >$ où :

- Q est un ensemble d'états.
- $T \subset Q \times A \times Q$ un ensemble de transitions.

Intuitivement, un système de transition formalise un processus :

- les états correspondent aux états du processus.
- l'alphabet A désigne les "actions" du processus.
- une transition (q, a, q') signifie que si le processus est dans l'état q, il peut exécuter l'action a et alors il passe dans l'état q'.

On considère maintenant un système de n processus $STA_i =< Q_i, T_i >$ sur A_i. Une manière de spécifier les interactions entre les processus d'un même système est d'énoncer des conditions de la forme:

- le processus STA_i doit effectuer l'action $a \in A_i$ pendant que le processus STA_j fait $b \in A_j$.

Ces interactions vont être exprimées dans un ensemble $S \subset A_1 \times A_2 ... \times A_n$ par des vecteurs de synchronisation :

- $\forall \vec{s} \in S$, $\prod_i \vec{s} = a$ *et* $\prod_j \vec{s} = b \Rightarrow$ le processus STA_i doit effectuer l'action $a \in A_i$ pendant que le processus STA_j fait $b \in A_j$.

L'ensemble des synchronisations peut être étendu à des conditions du type :

- le processus STA_i fait $a \in A_i$ avant (ou après) que le processus STA_j ne fasse $b \in A_j$.

3.1.2 Notre Modèle

Notre modèle est une extension du système précédent. Chaque état d'un processus est soit :

- une action à laquelle est associé un automate représentant son exécution.
- une structure de contrôle à laquelle est associé un automate (conditionnelle, itération, alternative, réactivité).
- une structure de contrôle parallèle à laquelle est associé un système de transition étendu.

Une transition (q, a, q') signifiera que si un processus est dans l'état q il passera dans l'état q' après l'exécution de l'automate a associé à q.

La synchronisation est quant à elle réduite aux états du processus représentant des actions se déroulant en parallèle. Elle sera modélisée par un graphe G. Les sommets de ce graphe représentent les différentes actions. L'ensemble des arcs est établi de la façon suivante :

- l'arc (a, b) est ajouté dans G si l'action b succède à l'action a dans un même processus.
- l'arc (a, b) est ajouté dans G s'il existe une règle de synchronisation du type : le processus STA_i doit exécuter l'action a avant que STA_j ne fasse l'action b.
- les arcs (a, b) et (b, a) sont ajoutés dans G s'il existe une règle de synchronisation du type : le processus STA_i doit effectuer l'action a pendant que STA_j exécute l'action b.

Le graphe G représente un ordonnancement des actions. Pour détecter un interblocage introduit par les règles de synchronisation, il suffira d'effectuer la recherche de certains cycles dans G. S'il existe un cycle sans paire d'arcs (a, b) (b, a), on pourra diagnostiquer un interblocage. Le graphe est composé :

- d'un ensemble de sommets $S = \{act_1, ..., act_n\}$.
- d'un ensemble d'arcs $A = \{(act_i, act_j), ...\}$ où $(act_i, act_j) \in S^2$.

Un ensemble de fonctions est associé à cette structure :

- $succ(G, act_i)$ donne l'ensemble des successeurs de act_i dans G.
- $pred(G, act_i)$ donne l'ensemble des prédecesseurs de act_i dans G.
- $supp(G, act_i)$ donne une copie du graphe G auquel on a supprimé l'action act_i.

Le graphe sera utilisé pour tester si une action peut être exécuté. Un sommet act_i du graphe de synchronisation peut être dans trois états :

- inactif : la possible exécution de l'action act_i n'a pas encore été testée.
- actif : l'état du graphe G a autorisé le lancement de l'action act_i.
- delay : l'état du graphe G n'autorise pas le lancement de l'action act_i, elle est alors mise en attente.

La fonction $delay(G)$ donne l'ensemble des sommets en attente du graphe G. Le prédicat $ready(G, act_i)$ est vrai si l'état du graphe G autorise le lancement de l'action act_i. Ce prédicat est évalué de la manière suivante :

Soit $E = pred(G, act_i)$
Si $\exists act_j \in E$ telque $act_j \notin succ(G, act_i)$ alors résultat faux.
Sinon on effectue l'itération sur E :
Soit $H = supp(G, act_i)$
Faire jusqu'à succès, échec
Quand $E = \emptyset$ sortir par succès
Soit act_j un élément de E
Quand $\neg ready(H, act_j)$ sortir par échec
$E := E \setminus \{act_j\}$
Refaire
fsi

3.2 La Sémantique

Nous présenterons pour chaque structure sa syntaxe :

- la représentation graphique qui est utilisée par l'opérateur pour créer le plan.
- la forme textuelle qui permet de conserver les plans et de les communiquer au contrôleur d'exécution (la grammaire de cette syntaxe est exposée *Appendix A*).

Nous énoncerons les règles qui définissent la sémantique de chaque structure. Elles seront soit des règles de transition de la forme :

$$< plan, \rho, G, \gamma > \xrightarrow[I]{ev} < plan', \rho', G', \gamma' >$$

Soit des règles d'inférence de déduction naturelle, de la forme :

$$\frac{< premisses... >}{< transition >}$$

Avec :

- ev : l'ensemble des signaux produit par le passage de $plan$ à $plan'$.
- I : l'environnement de signaux dans lequel s'est déroulée l'exécution.

- ρ : la mémoire de paramètres. Cette mémoire se présente sous la forme d'une liste d'ensembles de paramètres.
  - La fonction $prem(\rho)$ permet d'accéder à l'ensemble se situant en tête de liste, cet ensemble représente les paramètres courants de l'exécution. De plus cette fonction affecte à chacun des paramètres la valeur de l'expression qui lui est associée.
  - La fonction $suite(\rho)$ donne en résultat la mémoire de paramètres ρ privée de son premier élément.
  - La fonction $ajout(param, \rho)$ donne en résultat la mémoire de paramètres constituée de ρ auquelle on a ajouté, en tête de liste, l'ensemble de paramètres *param*.
- G : une liste de graphes de synchronisation avec $prem(G)$: graphe de synchronisation associé à la structure de contrôle parallèle courante.
- γ : la liste des ensembles d'actions continues lancées depuis le début de l'exécution d'une branche d'un plan. A la fin du parallélisme englobant cette branche, toutes les actions continues lancées, lors de l'exécution de ses branches, seront stoppées.

Un signal est noté $S(val)$ où S est le nom du signal et *val* la liste des valeurs qui lui sont associées. L'environnement dans lequel se trouve présent le signal S avec la valeur *val* est noté $S^+(val)$.

L'instruction *nothing* est l'action nulle. La règle de transition qui lui est associée est :

$$< nothing, \rho, G, \gamma > \xrightarrow[I]{\emptyset} \rho, G, \gamma$$

Nous saurons exécuter un plan écrit en YALTA quand nous posséderons les règles nécessaires pour réécrire totalement le plan.

$$< plan, \rho, G, \gamma > \xrightarrow{*} \rho', G', \gamma'$$

3.3 Les Expressions

Les expressions sont les objets qui vont permettre d'associer des valeurs aux paramètres des différentes actions. Une expression sera soit une constante, soit un paramètre, soit une donnée, soit une combinaison d'expressions à l'aide d'opérateurs.

Une constante C sera représentée par sa valeur :

$$[C]\rho = val(C)$$

Un paramètre sera représenté par le symbole qui lui est associé. Sa valeur sera lue dans la mémoire de paramètres ρ :

$$[X]\rho = prem(\rho).X$$

Quand aux données, elles sont représentées par leur symbole précédé d'une "*".

Le signal $Qval(symb)$ sera émis pour provoquer la consultation, par la base de connaissances, de la valeur associée à *symb* :

$$[*symb] \xrightarrow[I]{Qval(symb)} [await\ symb]$$

Lors de la réception de $Rval(symb, val)$, *symb* sera remplacé par sa valeur *val* :

$$[await\ symb] \xrightarrow[Rval^+(symb,val)]{\emptyset} [val]$$

On peut aussi combiner les expressions à l'aide d'opérateurs (addition, multiplication, ...) :

$$[exp_1\ op\ exp_2]\rho = ([exp_1]\rho)\ op\ ([exp_2]\rho)$$

3.4 Les Structures de Contrôle Informatiques

Nous allons présenter les structures de contrôle du langage liées à l'informatique. Ces structures sont présentes dans la plupart des langages informatiques : séquence, conditionnelle, itération, plus le parallélisme.

3.4.1 La Séquence

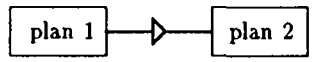

Figure 2 : Représentation graphique d'une séquence d'actions

La séquence est l'opérateur qui va permettre de spécifier si deux plans se succèdent. Pour clarifier la présentation nous nous limitons au cas de deux plans en séquence. La représentation interne de la séquence est :

$$(seq \quad plan_1 \; plan_2 \; ...)$$

[Seq 1]

L'exécution de $plan_1$ évolue en premier :

$$\frac{< plan_1, \rho, G, \gamma > \underset{I}{\overset{ev}{\longrightarrow}} < plan_1', \rho, G', \gamma' >}{< (seq \quad plan_1 \; plan_2) \; , \rho, G, \gamma > \underset{I}{\overset{ev}{\longrightarrow}} < (seq \quad plan_1' \; plan_2) \; , \rho, G', \gamma' >}$$

[Seq 2]

Celle de $plan_2$ sera lancée lorsque $plan_1$ se terminera :

$$\frac{< plan_1, \rho, G, \gamma > \underset{I}{\overset{ev}{\longrightarrow}} \rho, G', \gamma'}{< (seq \quad plan_1 \; plan_2) \; , \rho, G, \gamma > \underset{I}{\overset{ev}{\longrightarrow}} < plan_2, \rho, G', \gamma' >}$$

3.4.2 La Conditionnelle

La conditionnelle permet d'associer une condition à un plan. Pour clarifier la présentation nous nous limitons au cas de la conditionnelle simple (du type : si alors sinon). La représentation interne de la conditionnelle est alors :

$$\begin{array}{lll} (cond & (condition \; cond_1) & plan_1 \\ & (condition \; T) & plan_2) \end{array}$$

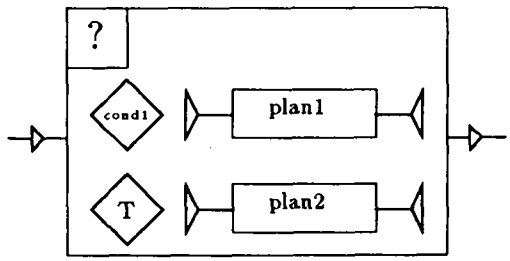

Figure 3 : Représentation graphique de la conditionnelle

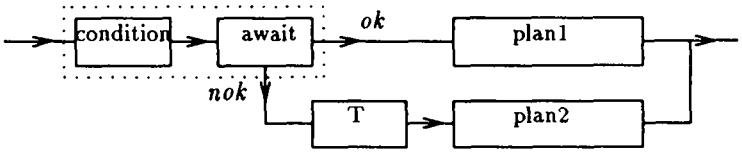

Figure 4 : Automate associé à l'exécution de la conditionnelle

[Cond 1]

La première condition est évaluée :

$$< \begin{matrix}(cond & (condition\ cond_1) & plan_1 \\ & (condition\ T) & plan_2)\end{matrix} , \rho, G, \gamma > \xrightarrow[I]{Qcond(cond_1, prem(\rho))}$$

$$< \begin{matrix}(cond & (condition\ cond_1\ await) & plan_1 \\ & (condition\ T) & plan_2)\end{matrix} , \rho, G, \gamma >$$

[Cond 2]

En cas de succès, l'instruction qui lui est associée est exécutée :

$$< \begin{matrix}(cond & (condition\ cond_1\ await) & plan_1 \\ & (condition\ T) & plan_2)\end{matrix} , \rho, G, \gamma > \xrightarrow[Rcond^+(cond_1, ok)]{\emptyset}$$

$$< plan_1, \rho, G, \gamma >$$

[Cond 3]

Sinon, on passe à l'évaluation de la condition suivante :

$$< \begin{matrix}(cond & (condition\ cond_1\ await) & plan_1 \\ & (condition\ T) & plan_2)\end{matrix} , \rho, G, \gamma > \xrightarrow[Rcond^+(cond_1, nok)]{\emptyset}$$

$$< (cond \quad (condition\ T) \quad plan_2) , \rho, G, \gamma >$$

[Cond 4]

Une branche "otherwise" existe par défaut :

$$< (cond \quad (condition\ T) \quad plan) , \rho, G, \gamma > \xrightarrow[I]{\emptyset} < plan, \rho, G, \gamma >$$

On lui associe éventuellement le plan vide.

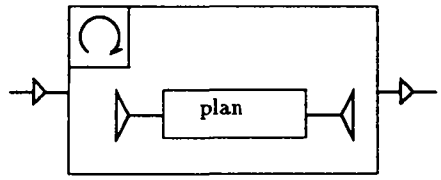

Figure 5 : Représentation graphique de l'itération

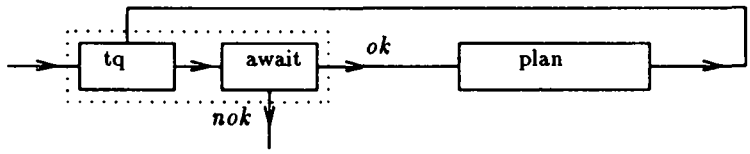

Figure 6 : Automate associé à l'exécution de l'itération

3.4.3 L'Itération

L'itération va permettre à l'opérateur de faire exécuter un plan tant qu'une condition est vérifiée. La représentation interne de l'itération est :

$$(tq \quad (condition\ cond_1) \quad plan)$$

[Tq 1]

Avant chaque exécution du plan, la condition est testée :

$$< (tq \quad (condition\ cond_1) \quad plan) \ , \rho, G, \gamma > \xrightarrow[I]{Qcond(cond_1, prem(\rho))} < (tq \quad (condition\ cond_1\ await) \quad plan) \ , \rho, G, \gamma >$$

[Tq 2]

En cas de succès le plan est lancé :

$$< (tq \quad (condition\ cond_1\ await) \quad plan) \ , \rho, G, \gamma > \xrightarrow[Rcond^+(cond_1, ok)]{\emptyset} < (seq \quad plan \quad (tq \quad (condition\ cond_1) \quad plan)) \ , \rho, G, \gamma >$$

[Tq 3]

Sinon l'itération est terminée :

$$< (tq \quad (condition\ cond_1\ await) \quad plan) \ , \rho, G, \gamma > \xrightarrow[Rcond^+(cond_1, nok)]{\emptyset} \rho, G, \gamma$$

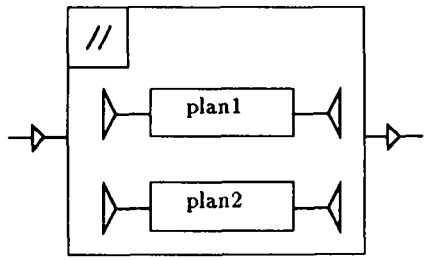

Figure 7 : Représentation graphique du parallélisme

3.4.4 Le Parallélisme

Le parallélisme est la structure de contrôle qui va permettre de spécifier que des plans peuvent s'exécuter en concurrence. Pour clarifier la présentation, nous nous limitons au cas de deux plans en parallèle. La représentation interne du parallélisme est :

$(par \quad état \quad G_{sync} \quad plan_1 \quad plan_2 \ ...)$ avec $état \in \{begin, \ exec\}$
et G_{sync} le graphe de synchronisation associé au parallélisme.

[Par 1]

Les exécutions de $plan_1$ et de $plan_2$ vont être lancées en parallèle :

$$< (par \quad begin \quad G_{sync} \quad plan_1 \quad plan_2) \ , \rho, G, \gamma > \xrightarrow[I]{\emptyset}$$

$$< (par \quad exec \quad G_{sync} \quad plan_1 \quad plan_2) \ , \rho, \ ajout(G_{sync}, G), \ ajout(\emptyset, \gamma) >$$

Le graphe de synchronisation du parallélisme G_{sync} est mis en tête de la liste G et l'ensemble des actions continues associées au parallélisme est initialisé à vide.

[Par 2]

Les exécutions de $plan_1$ et de $plan_2$ sont concurrentes :

$$\frac{< plan_1, \rho, G, \gamma > \xrightarrow[I]{ev} < plan_1', \rho, G', \gamma' >}{< (par \quad exec \quad G_{sync} \quad plan_1 \quad plan_2) \ , \rho, G, \gamma > \xrightarrow[I]{ev} < (par \quad exec \quad G_{sync} \quad plan_1' \quad plan_2) \ , \rho, \begin{array}{c} ajout(prem(G) \cap \\ prem(G'), suite(G)) \end{array} , \begin{array}{c} ajout(prem(\gamma) \cup \\ prem(\gamma'), suite(\gamma)) \end{array} >}$$

A chaque pas d'exécution on supprime de $prem(G)$ les actions qui ont été terminées (intersecton) et on ajoute à $prem(\gamma)$ les actions continues qui ont été lancées (union).

[Par 3]

$$\frac{< plan_2, \rho, G, \gamma > \xrightarrow[I]{ev} < plan_2', \rho, G', \gamma' >}{< (par \quad exec \quad G_{sync} \quad plan_1 \quad plan_2) \ , \rho, G, \gamma > \xrightarrow[I]{ev} < (par \quad exec \quad G_{sync} \quad plan_1 \quad plan_2') \ , \rho, \begin{array}{c} ajout(prem(G) \cap \\ prem(G'), suite(G)) \end{array} , \begin{array}{c} ajout(prem(\gamma) \cup \\ prem(\gamma'), suite(\gamma)) \end{array} >}$$

[Par 4]

Lorsqu'une instruction se termine elle est supprimée du parallélisme :

$$\frac{< plan_1, \rho, G, \gamma > \overset{ev}{\underset{I}{\longrightarrow}} \rho, G', \gamma'}{< (par \quad exec \quad G_{sync} \quad plan_1 \quad plan_2) \ , \rho, G, \gamma > \overset{ev}{\underset{I}{\longrightarrow}} < (par \quad exec \quad G_{sync} \quad plan_2) \ , \rho, \begin{matrix} ajout(prem(G) \cap \\ prem(G'), suite(G)) \end{matrix} \ , \ \begin{matrix} ajout(prem(\gamma) \cup \\ prem(\gamma'), suite(\gamma)) \end{matrix} >}$$

[Par 5]

$$\frac{< plan_2, \rho, G, \gamma > \overset{ev}{\underset{I}{\longrightarrow}} \rho, G', \gamma'}{< (par \quad exec \quad G_{sync} \quad plan_1 \quad plan_2) \ , \rho, G, \gamma > \overset{ev}{\underset{I}{\longrightarrow}} < (par \quad exec \quad G_{sync} \quad plan_1) \ , \rho, \begin{matrix} ajout(prem(G) \cap \\ prem(G'), suite(G)) \end{matrix} \ , \ \begin{matrix} ajout(prem(\gamma) \cup \\ prem(\gamma'), suite(\gamma)) \end{matrix} >}$$

[Par 6]

Le parallélisme est terminé lorsque toutes ses branches le sont. A la fin du parallélisme, toutes les actions continues lancées, lors de l'exécution de ses branches, sont stoppées :

$$\frac{< plan, \rho, G, \gamma > \overset{ev}{\underset{I}{\longrightarrow}} \rho, G', \gamma'}{< (par \quad exec \quad G_{sync} \quad plan) \ , \rho, G, \gamma > \overset{\begin{matrix} ev \times \{\forall act_i \in prem(\gamma') \\ Qact(act_i, end)\} \end{matrix}}{\underset{I}{\longrightarrow}} \rho, \ suite(G), \ suite(\gamma)}$$

3.5 Les Structures de Contrôle liées à la Téléopération

Elles correspondent à l'exécution d'une action, à l'intervention de l'opérateur et à la prise en compte des évènements de l'environnement durant l'exécution d'un plan.

3.5.1 Les Actions

Qu'elle soit atomique, composée ou continue, une action est représentée graphiquement par une boite contenant soit le nom de sa classe, soit l'icône associée à sa classe. La représentation interne d'une action est :

$(act \quad type \quad nom\_instance \quad état)$ avec $type \in \{ActAtom, \ ActComp, \ ActCont\}$ et $état \in \{nil, \ delay, \ await, \ exec, \ final, \ trap\}$

Une action atomique permet à l'opérateur de spécifier un ordre directement interprété par le robot.

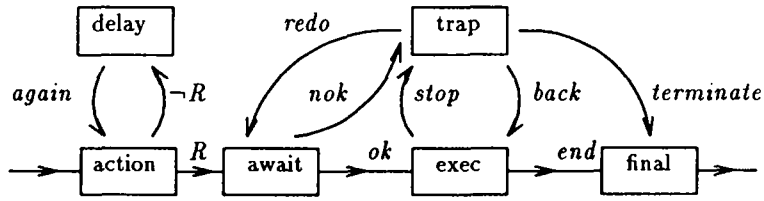

Figure 8 : Automate associé à l'exécution d'une action atomique

[ActAtom 1]

Si l'action est prête à être exécutée, on émet un signal pour déclencher la vérification de ses préconditions :

$$\frac{ready(prem(G), act_i)}{< (act \quad ActAtom \quad act_i \quad nil) \ , \rho, G, \gamma > \overset{\overset{Qact(act_i, pre, prem(\rho))}{\longrightarrow}}{I} < (act \quad ActAtom \quad act_i \quad await) \ , \rho, G, \gamma >}$$

[ActAtom 2]

Sinon l'action est mise en attente :

$$\frac{\neg ready(prem(G), act_i)}{< (act \quad ActAtom \quad act_i \quad nil) \ , \rho, G, \gamma > \overset{\overset{\emptyset}{\longrightarrow}}{I} < (act \quad ActAtom \quad act_i \quad delay) \ , \rho, G, \gamma >}$$

[ActAtom 3]

Si la vérication des préconditions est un succès, le signal nécessaire au déclenchement de l'exécution de act_i est émis :

$$< (act \quad ActAtom \quad act_i \quad await) \ , \rho, G, \gamma > \overset{\overset{Qact(act_i, start, prem(\rho))}{\longrightarrow}}{Ract^+(act_i, ok)} < (act \quad ActAtom \quad act_i \quad exec) \ , \rho, G, \gamma >$$

[ActAtom 4]

Sinon, l'exécution de l'action est mise dans un état "trap" et une alarme informant de la non vérification des préconditions est émise :

$$< (act \quad ActAtom \quad action_i \quad await) \ , \rho, G, \gamma > \overset{\overset{Qalarm(act_i, pre)}{\longrightarrow}}{Ract^+(act_i, nok)} < (act \quad ActAtom \quad act_i \quad trap) \ , \rho, G, \gamma >$$

[ActAtom 5]

Le signal $Ract(act_i, end)$ annonce l'achèvement de l'action :

$$< (act \quad ActAtom \quad act_i \quad exec) \ , \rho, G, \gamma > \quad \overset{\emptyset}{\underset{Ract^+(act_i, end)}{\longrightarrow}} \quad < (act \quad ActAtom \quad act_i \quad final) \ , \rho, G, \gamma >$$

[ActAtom 6]

Le signal $Ract(act_i, stop)$ informe de la demande d'interruption de l'exécution de act_i. L'action est mise dans un état "trap" et une alarme prévenant de l'interruption est émise :

$$< (act \quad ActAtom \quad act_i \quad exec) \ , \rho, G, \gamma > \quad \overset{Qalarm(act_i, interrupt)}{\underset{Ract^+(act_i, stop)}{\longrightarrow}} \quad < (act \quad ActAtom \quad act_i \quad trap) \ , \rho, G, \gamma >$$

[ActAtom 7]

Après son exécution, l'action est supprimée du graphe de synchronisation courant. Des signaux sont émis vers toutes les actions en attente de ce graphe pour les prévenir de la fin d'une action. Le signal $Qact(act_i, effet, prem(\rho))$ provoque la prise en compte, par la base de connaissances, des effets de l'action.

$$< (act \quad ActAtom \quad act_i \quad final) \ , \rho, G, \gamma > \quad \overset{\begin{array}{c} Qact(act_i, effet, prem(\rho)) \times \\ \{\forall act_j \in delay(prem(G)) \ \ Ract(act_j, again)\} \end{array}}{\underset{I}{\longrightarrow}} \quad \rho, \ ajout(supp(prem(G), act_i), suite(G)), \ \gamma$$

[ActAtom 8]

Le signal $Ract(act_i, redo)$ transmet la demande de refaire une tentative d'exécution complète de l'action :

$$< (act \quad ActAtom \quad act_i \quad trap) \ , \rho, G, \gamma > \quad \overset{\emptyset}{\underset{Ract^+(act_i, redo)}{\longrightarrow}} \quad < (act \quad ActAtom \quad act_i \quad nil) \ , \rho, G, \gamma >$$

[ActAtom 9]

Le signal $Ract(act_i, back)$ prévient de la demande de reprise de l'exécution de l'action. $Qact(act_i, back)$ permet de relancer l'action à compter de son point d'interruption :

$$< (act \quad ActAtom \quad act_i \quad trap) \ , \rho, G, \gamma > \quad \overset{Qact(act_i, back, prem(\rho))}{\underset{Ract^+(act_i, back)}{\longrightarrow}} \quad < (act \quad ActAtom \quad act_i \quad exec) \ , \rho, G, \gamma >$$

[ActAtom 10]

Le signal $Ract(act_i, terminate)$ signifie que l'action a été exécutée d'une manière différente, elle est alors considérée terminée :

$$< (act \quad ActAtom \quad act_i \quad trap) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, terminate)]{\emptyset} < (act \quad ActAtom \quad act_i \quad final) \ , \rho, G, \gamma >$$

[ActAtom 11]

Le signal $Ract(act_i, again)$ informe de l'achèvement d'une action et donc de la modification du graphe de synchronisation. L'action est enlevée de l'état d'attente :

$$< (act \quad ActAtom \quad act_i \quad delay) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, again)]{\emptyset} < (act \quad ActAtom \quad act_i \quad nil) \ , \rho, G, \gamma >$$

Une action continue va permettre à l'opérateur de lancer une action atomique sans s'occuper de sa terminaison. Une action succédant à une action continue, dans un plan, est exécutée sans attendre la fin de l'action continue. Une action continue est stoppée lorsque le bloc parallèle l'englobant se termine. Les effets de l'action sont pris en compte au cours de son exécution.

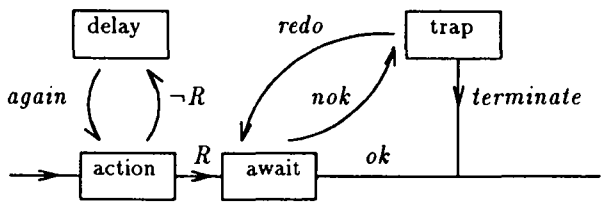

Figure 9 : Automate associé à l'exécution d'une action continue

[ActCont 1]

Si l'action est prête à être exécutée, on émet un signal pour déclencher la vérification de ses préconditions :

$$\frac{ready(prem(G), act_i)}{< (act \quad ActCont \quad act_i \quad nil) \ , \rho, G, \gamma > \xrightarrow[I]{Qact(act_i, pre, prem(\rho))} < (act \quad ActCont \quad act_i \quad await) \ , \rho, G, \gamma >}$$

[ActCont 2]

Sinon l'action est mise en attente :

$$\frac{\neg ready(prem(G), act_i)}{< (act \quad ActCont \quad act_i \quad nil) \ , \rho, G, \gamma > \xrightarrow[I]{\emptyset} < (act \quad ActCont \quad act_i \quad delay) \ , \rho, G, \gamma >}$$

[ActCont 3]

Si la vérication des préconditions est un succès, le signal nécessaire au déclenchement de l'exécution de l'action est émis. L'action est supprimée du graphe de synchronisation courant. Des signaux sont émis vers toutes les actions en attente de ce graphe. L'action est de plus ajoutée à la liste des actions continues en cours :

$$< (act \quad ActCont \quad act_i \quad await) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, ok)]{\begin{array}{c} Qact(act_i, start, prem(\rho)) \times \\ \{\forall act_j \in delay(prem(G)) \ \ Ract(act_j, again)\} \end{array}}$$

$$\rho, \ ajout(supp(prem(G), act_i), suite(G)), \ ajout(\{act_i\} \cup prem(\gamma)), suite(\gamma))$$

[ActCont 4]

Sinon, l'exécution de l'action est mise dans un état "trap" et une alarme prévenant de la non vérification des préconditions est émise :

$$< (act \quad ActCont \quad act_i \quad await) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, nok)]{Qalarm(act_i, pre)}$$

$$< (act \quad ActCont \quad act_i \quad trap) \ , \rho, G, \gamma >$$

[ActCont 5]

Le signal $Ract(act_i, redo)$ prévient de la demande de refaire une tentative pour lancer l'exécution de l'action :

$$< (act \quad ActCont \quad act_i \quad trap) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, redo)]{\emptyset}$$

$$< (act \quad ActCont \quad act_i \quad nil) \ , \rho, G, \gamma >$$

[ActCont 6]

Le signal $Ract(act_i, terminate)$ signifie que l'action a été lancée d'une manière différente :

$$< (act \quad ActCont \quad act_i \quad trap) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, terminate)]{\{\forall act_j \in delay(prem(G)) \ \ Ract(act_j, again)\}}$$

$$\rho, \ ajout(supp(prem(G), act_i), suite(G)), \ ajout(\{act_i\} \cup prem(\gamma)), suite(\gamma))$$

[ActCont 7]

Le signal $Ract(act_i, again)$ provoque le passage de l'action de l'état *en attente* à l'état *exécutable* :

$$< (act \quad ActCont \quad act_i \quad delay) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, again)]{\emptyset}$$

$$< (act \quad ActCont \quad act_i \quad nil) \ , \rho, G, \gamma >$$

Les actions composées vont permettre à l'opérateur d'associer à une action : un plan, des paramètres, des préconditions, des règles et des effets. Ces actions permettent de hiérarchiser les plans et de créer des plans types. Une action composée est exécutée dans l'environnement de paramètres restreint à ses propres paramètres. La visibilité d'un paramètre est limitée à l'action composée ou au plan où il est défini, il n'y a donc pas de liaison dynamique entre actions composées.

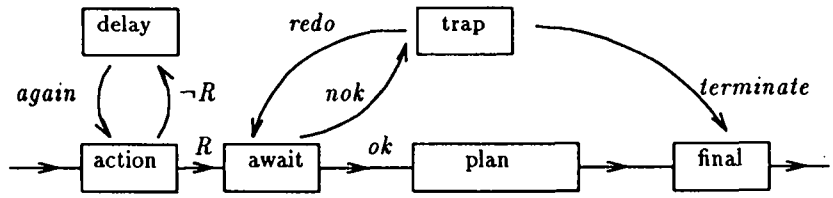

Figure 10 : Automate associé à l'exécution d'une action composée

[ActComp 1]

Si l'action est prête à être exécutée, on émet un signal pour déclencher la vérification de ses préconditions :

$$ready(prem(G), act_i)$$

$$< (act \quad ActComp \quad act_i \quad nil) \ , \rho, G, \gamma > \quad \overset{Qact(act_i, pre, prem(\rho))}{\underset{I}{\longrightarrow}} \quad < (act \quad ActComp \quad act_i \quad await) \ , \rho, G, \gamma >$$

[ActComp 2]

Sinon l'action est mise en attente :

$$\neg ready(prem(G), act_i)$$

$$< (act \quad ActComp \quad act_i \quad nil) \ , \rho, G, \gamma > \quad \overset{\emptyset}{\underset{I}{\longrightarrow}} \quad < (act \quad ActComp \quad act_i \quad delay) \ , \rho, G, \gamma >$$

[ActComp 3]

Si la vérication des préconditions est un succès, le signal nécessaire au déclenchement de l'exécution est émis. Le corps de l'action est ajouté au programme. L'ensemble des paramètres locaux de l'exécution est mis en tête de la liste ρ :

$$< (act \quad ActComp \quad act_i \quad await) \ , \rho, G, \gamma > \quad \overset{\emptyset}{\underset{Ract^+(act_i, ok, param, plan)}{\longrightarrow}} \quad < (seq \quad plan \quad (act \quad ActComp \quad act_i \quad final) \ ) \ , ajout(param, \rho), G, \gamma >$$

[ActComp 4]

Sinon, l'exécution de l'action est mise dans un état "trap" et une alarme prévenant de la non vérification des préconditions de l'action est émise :

$$< (act \quad ActComp \quad act_i \quad await) \ , \rho, G, \gamma > \quad \overset{Qalarm(act_i, pre)}{\underset{Ract^+(act_i, nok)}{\longrightarrow}} \quad < (act \quad ActComp \quad act_i \quad trap) \ , \rho, G, \gamma >$$

[ActComp 5]

L'action est supprimée du graphe de synchronisation courant. Des signaux sont émis vers toutes les actions en attente de ce graphe. Le signal $Qact(act_i, effet, prem(\rho))$ provoque la prise en compte des effets de l'action, par la base de connaissances. La liste des paramètres locaux est supprimée de γ :

$$< (act \quad ActComp \quad act_i \quad final) \ , \rho, G, \gamma > \xrightarrow[I]{\substack{Qact(act_i, effet, prem(suite(\rho))) \times \\ \{\forall act_j \in delay(prem(G)) \ \ Ract(act_j, again)\}}}$$

$$suite(\rho), \ \ ajout(supp(prem(G), act_i), suite(G)), \ \ \gamma$$

[ActComp 6]

Le signal $Ract(act_i, redo)$ informe de la demande de refaire une tentative d'exécution complète de l'action :

$$< (act \quad ActComp \quad act_i \quad trap) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, redo)]{\emptyset}$$

$$< (act \quad ActComp \quad act_i \quad nil) \ , \rho, G, \gamma >$$

[ActComp 7]

Le signal $Ract(act_i, terminate)$ signifie que l'action a été exécutée d'une manière différente, elle est alors considérée terminée :

$$< (act \quad ActComp \quad act_i \quad trap) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, terminate)]{\emptyset}$$

$$< (act \quad ActComp \quad act_i \quad final) \ , ajout(\emptyset), \rho, G, \gamma >$$

[ActComp 8]

Le signal $Ract(act_i, again)$ provoque le passage de l'action de l'état *en attente* à l'état *exécutable* :

$$< (act \quad ActComp \quad act_i \quad delay) \ , \rho, G, \gamma > \xrightarrow[Ract^+(act_i, again)]{\emptyset}$$

$$< (act \quad ActComp \quad act_i \quad nil) \ , \rho, G, \gamma >$$

3.5.2 La Réactivité

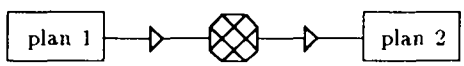

Figure 11 : Représentation graphique de la réactivité

La réactivité va donner la possibilité à l'opérateur de synchroniser un point précis du plan sur un évènement. Cet évènement peut être produit par l'opérateur aussi bien que par l'environnement de l'application. Sa représentation interne est :

$$(react \quad event_i \quad état) \quad \text{avec } état \in \{nil, \ await\}$$

[React 1]

Lorsque l'exécution du plan arrive au niveau d'une réactivité, l'exécution de cette branche se met en attente de l'évènement associé. Un signal est envoyé vers l'opérateur pour lui signifier la mise en sommeil d'une branche, et le type de l'évènement attendu :

$$< (react \quad ev_i) \ , \rho, G, \gamma > \xrightarrow[I]{Qevent(ev_i, prem(\rho))} < (react \quad ev_i \quad await) \ , \rho, G, \gamma >$$

[React 2]

L'exécution de la branche reprend son cours lors de la première occurence de l'évènement :

$$< (react \quad ev_i \quad await) \ , \rho, G, \gamma > \xrightarrow[Revent^{+}(ev_i)]{\emptyset} \rho, G, \gamma$$

3.5.3 L'Alternative

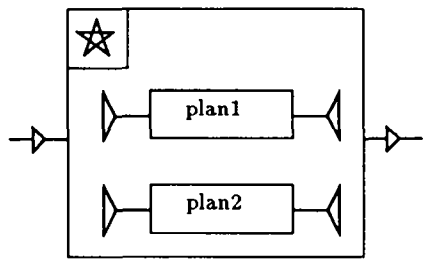

Figure 12 : Représentation graphique de l'alternative

Cette structure de contrôle permet d'associer à une instruction du plan, un ensemble d'alternatives. Chaque alternative correspond soit à un plan, soit à une action. Pour une question de clarté, nous nous limitons au cas de l'alternative entre deux plans. La représentation interne est :

$$(alt \quad état \quad num \quad plan_1 \ plan_2 \ ...) \quad \text{avec } état \in \{begin, \ await\}$$

L'alternative se différencie de la conditionnelle, dans la mesure où la raison de l'exécution de l'une des alternatives n'est pas modélisable par une condition. Lors de l'exécution de l'alternative, le choix de la branche à exécuter est fait par l'opérateur.

[Alt 1]

Le signal $Qalt(num)$ prévient l'opérateur de l'exécution de l'alternative num :

$$< (alt \quad begin \quad num \quad plan_1 \ plan_2) \ , \rho, G, \gamma > \xrightarrow[I]{Qalt(num)} < (alt \quad await \quad num \quad plan_1 \ plan_2) \ , \rho, G, \gamma >$$

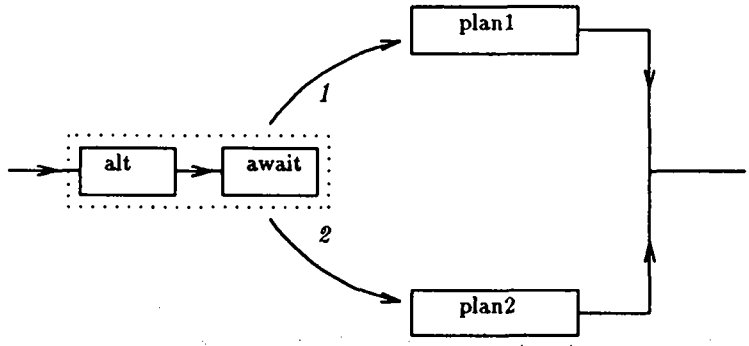

Figure 13 : Automate associé à l'exécution de l'alternative

[Alt 2]

Le signal $Ralt(num, i)$ signifie que l'opérateur a choisi l'exécution de la branche i :

$$< (alt \quad await \quad num \quad plan_1 \; plan_2) \; , \rho, G, \gamma > \xrightarrow[Ralt^+(num,i)]{\emptyset} < plan_i, \rho, G, \gamma >$$

Nous venons de présenter l'ensemble des structures de contrôle de YALTA, ainsi que leur sémantique. Cet ensemble contient l'ensemble minimal des structures composant tout langage informatique, plus une diversité d'actions : atomique, continue et composée, et les notions : alternative et réactivité, permettant à un opérateur de créer et de contrôler l'exécution de plans pour une application téléopérée. YALTA tient donc compte à la fois de l'enchaînement automatique des actions et de la présence active d'un opérateur durant leur exécution. Nous savons à présent exécuter un programme écrit en YALTA puisque savoir exécuter un programme équivaut à savoir le réécrire jusqu'à ce qu'on ne le puisse plus, or nous venons de voir comment réécrire toutes les structures de contrôle du langage.

4 Exemple de Plan

Avant de conclure, nous présentons deux exemples de plans modélisés à l'aide de YALTA. Le premier montre une programmation d'un plan pour une application réaliste : l'assemblage de connecteurs. Le

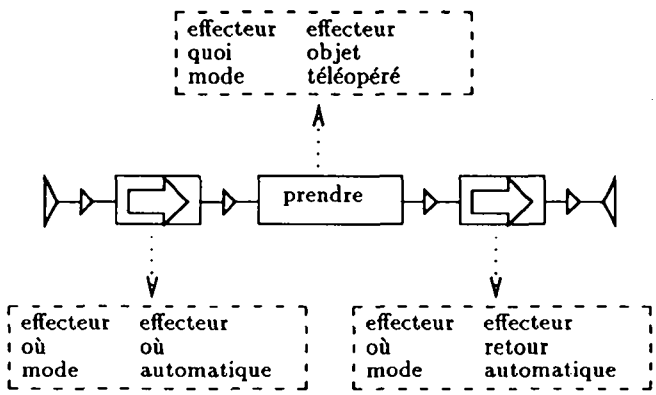

Figure 14 : Représentation graphique de l'action composée chercher

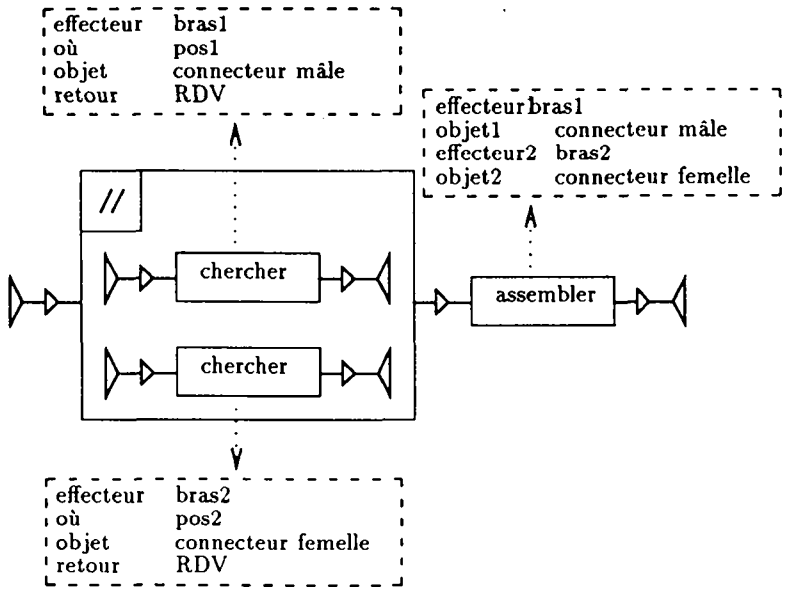

Figure 15 : Représentation graphique du plan exemple 1

second illustre le fonctionnement de la synchronisation.

4.1 Assemblage de Connecteurs

Ce plan est inspiré d'une étude, pour un système bi-bras, faite à Matra Espace [6]. Il décrit l'assemblage d'un connecteur : le premier bras va chercher un connecteur mâle, parallèlement le second bras va chercher un connecteur femelle, les deux bras se rejoignent en un lieu de rendez-vous pour effectuer l'assemblage. *Figures 15, 14 et 16*, nous donnons la représentation graphique du plan (et des actions composées qui le constituent) ainsi que le détail des passages de paramètres.

Nous associons au plan et aux actions les paramètres suivants :

Le plan :

- *bras1* : premier bras de l'application.
- *bras2* : second bras de l'application.
- *pos1* : position du connecteur mâle.
- *pos2* : position du connecteur femelle.
- *RDV* : lieu de rendez-vous pour l'assemblage des connecteurs.

Chercher :

- *effecteur* : effecteur qui va aller chercher l'objet.
- *où* : position de l'objet.
- *objet* : nom de l'objet.
- *retour* : lieu où ramener l'objet.

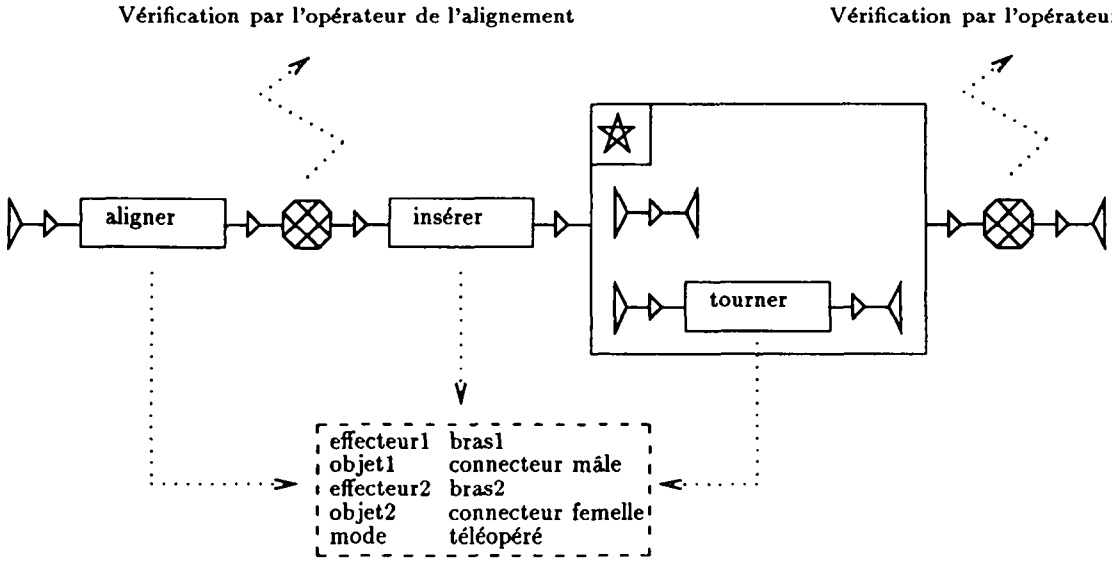

Figure 16 : Représentation graphique de l'action composée assembler

Assembler, Aligner, Insérer, Tourner :

- *effecteur1* : effecteur tenant le premier objet.
- *objet1* : nom du premier objet.
- *effecteur2* : effecteur tenant le second objet.
- *objet2* : nom du second objet.
- *mode* : mode d'exécution de l'action.

Pour les actions suivantes, *prendre* et *aller*, nous allons en plus spécifier les valeurs associées aux champs *préconditions*, *règles* et *effets*. L'action *aller* est modélisée graphiquement par une icône représentant une flèche.

Aller :

- *effecteur* : effecteur qui se déplace.
- *où* : position finale du déplacement.
- *mode* : mode d'exécution de l'action.
- *préconditions* : l'effecteur n'est pas entrain d'effectuer une action incompatible avec un déplacement.
- *règles* : existence permanente d'une distance de sécurité entre l'effecteur et les éléments de l'environnement.
- *effets* : la position finale est atteinte par l'effecteur.

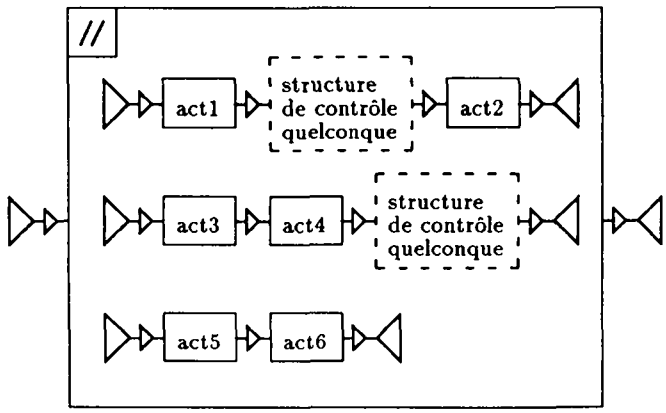

Figure 17 : Plan Exemple 2

Prendre :

- *effecteur* : effecteur qui doit se saisir de l'objet.
- *quoi* : nom de l'objet à prendre.
- *mode* : mode d'exécution de l'action.
- *préconditions* : l'effecteur est libre, l'objet est présent, l'effecteur peut prendre un objet de ce type.
- *effets* l'effecteur tient l'objet, l'objet est tenu par l'effecteur, la position de l'objet et celle de l'effecteur sont à présent liées.

4.2 Synchronisation

Le plan *figure 17* illustre l'exécution de trois branches en parallèle. Les règles de synchronisation sont les suivantes :

- act_2 et act_4 doivent être lancées simultanément.
- act_5 doit être lancé après l'exécution de act_1.
- act_6 doit être lancé après l'exécution de act_4.

Le graphe de synchronisation est donné *figure 18*. Lors de l'exécution du plan act_1 et act_3 pourront être lancées alors que act_5 sera mise en attente. act_5 ne sera lancée que si act_1 est terminée.

5 Conclusion

L'orientation principale de notre travail est l'amélioration des langages de programmation pour des applications en télérobotique. Le langage que nous avons défini s'apparente à un langage de commande, par ses structures de contrôle, son système de passage de paramètres et ses domaines d'applications.

Un de nos objectifs restera de fournir à l'opérateur un outil de représentation de plan qui sera graphiquement clair et explicite, aussi bien durant la phase de spécification, que durant la phase d'exécution. Si un accident survient durant l'exécution d'un plan, l'opérateur voudra savoir rapidement où l'erreur est intervenue et où reprendre l'exécution. Le langage devra être assez souple pour permettre la modification en ligne du plan, et assez puissant pour effectuer un contrôle durant l'exécution.

Nous venons de présenter le langage YALTA et le système dans lequel il doit s'intégrer. Cette étude fait suite à des bibliographies sur les différents systèmes de contrôle d'exécution [15] et sur les outils graphiques aptes à le modéliser [9]. Nous avons spécifié l'aspect visuel et la sémantique du langage.

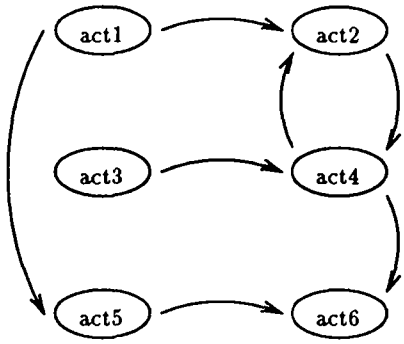

Figure 18 : Graphe de Synchronisation

La structure graphique du langage permettra à l'opérateur de créer facilement un plan d'actions pour son application, d'avoir un retour visuel de l'exécution de son plan, et surtout de pouvoir en permanence agir sur l'exécution. Cette dernière possibilité équivaut à effectuer un contrôle manuel de l'exécution.

Nous introduisons dans notre système un contrôle automatique. Ce contrôle s'effectue durant l'exécution grâce aux préconditions, règles et effets associés à chaque classe d'action, lesquels permettent le fonctionnement d'un module d'inférence et la mise à jour de sa base de connaissances.

Nous étudierons le module d'interprétation du langage. Ce module a pour fonctionnalité l'exécutabilité d'un plan conçu à l'aide de YALTA. L'exécution pourra se faire par réécriture successive de l'automate ou du plan. Une réécriture peut être envisagée à partir de règles de transition telles que nous les avons exposées. Il ne faut surtout pas perdre de vue les possibles modifications du plan en cours d'exécution.

Appendix A

Grammaire de la syntaxe interne de YALTA

S ::= (**seq** Liste_Instructions)

Liste_Instructions ::= Instruction ||
Instruction Liste_Instructions

Instruction ::= (**alt** Etat_Alt **num** Liste_Branches) ||
(**par** Etat_Par G_sync Liste_Branches) ||
(**cond** Liste_Branches_Cond) ||
(**act** Type_Action Nom_Action Etat_Action) ||
(**react** Nom_Action Etat_React) ||
(**tq** Condition S) ||
nothing

Liste_Branches ::= S ||
S Liste_Branches

Liste_Branches_Cond ::= Condition S ||
Condition S Liste_Branches_Cond

Condition ::= (**condition** Nom_Condition Etat_Condition)

Type_Action ::= **Act_Atom** || **Act_Cont** || **Act_Comp**

Etat_Action ::= **nil** || **await** || **exec** || **final** || **trap** || **delay**

Etat_Alt ::= **begin** || **await**

Etat_Par ::= **begin** || **exec** || **final**

Etat_React ::= **nil** || **await**

Etat_Condition ::= **nil** || **await**

Nom_Action ::= $\mathbf{act}_1$ || ... || $\mathbf{act}_i$

Nom_Condition ::= $\mathbf{cond}_1$ || ... || $\mathbf{cond}_j$ || **T**

Nom_Evenement ::= $\mathbf{ev}_1$ || ... || $\mathbf{ev}_k$

G_sync ::= $\mathbf{G}_1$ || ... || $\mathbf{G}_l$

Bibliographie

[1] B. BESOMBES, B. JULLIEN, P. LADET. *SAGASSE : Système expert pour l'aide à la gestion d'atelier flexible.* Ecole des Mines Saint Etienne, Rapport 87/12/BB.

[2] R. VALETTE. *Nets in production systems.* in Lecture notes in computer science, 255, Advances in Petri nets, September 1986.

[3] L. MARCE. *Contribution à l'autonomie des robots mobiles.* Thèse d'état, Rennes, Juillet 1984.

[4] EUREKA. *AMR : Advanced Mobile Robots.* Phase1, mid-term report, Juin 1987.

[5] G. ANDRE, T. BLAIS. *Space teleoperation and control concept, experimental evaluation for the Hermes robot arm (HERA).* in C.A. Manson, editor, Procedings of the international symposium on teleoperation and control, Bristol, England, July 1988.

[6] G. ANDRE. Communications personnelles, 1989.

[7] P. MIGAUD. *Contrôle d'exécution de plan d'actions pour un robot mobile.* thèse, UPS Toulouse, Novembre 1986.

[8] P. GRAVEZ. *Etude d'un système de supervision pour la téléopération assistée par ordinateur.* thèse, UST Lille, Mars 1988.

[9] J.C. PAOLETTI, L. MARCE. *Quelques outils graphiques pour la modélisation du contrôle d'exécution en robotique de coopération.* Publication interne, 475, Irisa/Inria Rennes, Juin 1989.

[10] E. RUTTEN, L. MARCE. *Plan simulation using temporal logics* in Proceedings of the IJCAI'89 workshop on integrated human-machine intelligence in aerospace systems, Detroit, Michigan, August 1989.

[11] G.D. PLOTKIN. *A structural approach to operationnal semantics* in lecture notes, Aarhus University, 1981.

[12] D. HAREL, A. PNUELI. *On the development of reactive systems* in Logic and Models of Concurrent Systems, Proceedings NATO, NATO ASI Series F, vol. 13, pp. 477-498, 1985.

[13] G. GONTHIER. *Sémantiques et modèles d'exécution des langages réactifs synchrones; application à Esterel* Thèse Paris Orsay, mars 1988.

[14] A. ARNOLD, M. NIVAT. *Comportement de processus* in Colloque AFCET "Les mathématiques de l'informatique", pp 35-68, 1982.

[15] C. JAUFFRINEAU *Contrôle d'exécution en robotique de coopération.* Publication interne, 416, Irisa/Inria Rennes, Juin 1988.

[16] E. RUTTEN, J.C. PAOLETTI, L. MARCE. *A task-level language for operator assistance in teleoperation.* to appear in Human-Machine Interaction and Artificial Intelligence in Aeronautics and Space, Toulouse, France, Septembre 1990.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 530 **SEMI-GRANULES AND SCHIELDING FOR OFF-LINE SCHEDULING**
Bernard LE GOFF, Paul LE GUERNIC, Julian ARAOZ DURAND
Avril 1990, 46 Pages.

PI 531 **DATA-FLOW TO VON NEUMANN : THE SIGNAL APPROACH**
Paul LE GUERNIC, Thierry GAUTIER
Avril 1990, 22 Pages.

PI 532 **OPERATIONAL SEMANTICS OF A DISTRIBUTED OBJECT-ORIENTED LANGUAGE AND ITS Z FORMAL SPECIFICATION**
Marc BENVENISTE
Avril 1990, 100 Pages.

PI 533 **ADAPTATION DE LA METHODE DE DAVIDSON A LA RESOLUTION DE SYSTEMES LINEAIRES : IMPLEMENTATION D'UNE VERSION PAR BLOCS SUR UN MULTIPROCESSEUR**
Miloud SADKANE, Brigitte VITAL
Avril 1990, 34 Pages.

PI 534 **DIFFUSE INTERREFLECTIONS. TECHNIQUES FOR FORM-FACTOR COMPUTATION**
Xavier PUEYO
Mai 1990, 28 Pages.

PI 535 **A NOTE ON GUARDED RECURSION**
Eric BADOUEL, Philippe DARONDEAU
Mai 1990, 10 Pages.

PI 536 **TOWARDS DOCUMENT ENGINEERING**
Vincent QUINT, Marc NANARD, Jacques ANDRE
Mai 1990, 20 Pages.

PI 537 **YALTA : YET ANOTHER LANGUAGE FOR TELEOPERATE APPLICATIONS**
Jean-Christophe PAOLETTI, Lionel MARCE
Juin 1990, 32 Pages.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique .

ISSN 0249-6399