HAL
open science

Architecture de contrôle distribuée pour robot mobile autonome : principes, conception et applications

Sara Fleury

To cite this version:

Sara Fleury. Architecture de contrôle distribuée pour robot mobile autonome : principes, conception et applications. Automatique / Robotique. Université Paul Sabatier - Toulouse III, 1996. Français. NNT : . tel-00165569

HAL Id: tel-00165569

https://theses.hal.science/tel-00165569v1

Submitted on 26 Jul 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Année 1996

Thèse

préparée au

Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS

en vue de l'obtention du

Doctorat de l'Université Paul Sabatier de Toulouse

Spécialité : Robotique

par

Sara FLEURY

Ingénieur de l'Ecole Nationale Supérieure d'Arts et Métiers

Architecture de contrôle distribuée pour robots mobiles autonomes: principes, conception et applications

---

Soutenue le 15 Fevrier 1996 devant le jury:

Président	**Georges**	**GIRALT**
Directeur de thèse	**Raja**	**CHATILA**
Rapporteurs	**Bernard**	**ESPIAU**
	Oussama	**KHATIB**
	Jean-Louis	**LACOMBE**
Examinateurs	**Marc**	**COURVOISIER**
	Malik	**GHALLAB**

Cette thèse a été préparée au LAAS-CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4

Rapport LAAS N° 96156

Avant Propos

Le travail présenté dans ce mémoire est avant tout le fruit d'une collaboration scientifique et humaine riche et chaleureuse que je dédie à mes très chers collègues et amis.

Je tiens à remercier Alain Costes qui a su conférer au Laboratoire d'Analyse et d'Architecture des Systèmes, par une animation professionnelle mais aussi extra-professionnelle, son caractère si particulier.

Un très grand merci à Georges Giralt pour m'avoir chaleureusement accueillie dans le groupe Robotique et Intelligence Artificielle, et pour son soutien attentionné tout au long de ces années. Je salue à cette occasion Malik Ghallab qui a pris sa relève à la direction de l'équipe et qui, par son énergie et sa grande disponibilité, en garantit la cohésion.

La thèse, objet de ce mémoire, a été réalisée sous l'œil avisé de Raja Chatila. Par sa pugnacité il a su maintenir le cap qui, au gré du flux et reflux des thèses, mène inexorablement vers l'Architecture-de-contrôle-décisionnelle-pour-robots-mobiles-autonomes. De façon plus personnelle je le remercie d'avoir su être présent lors des passages difficiles... en particulier lors de la dernière étape.

Ces travaux ont pu être mis en valeur dans le cadre d'un projet ambitieux de coordination multi-robots orchestré par Rachid Alami. Son enthousiasme et son investissement personnel ont imprégné toute l'équipe "STRADA". Merci à lui et à toute la robot-circus-band: Fred, Felix, Maher, Hanna, Luis, Nic, Matthieu, pour les moments intenses que nous avons vécu ensemble.

Je glisse ici un clin d'œil tout particulier à Matthieu avec qui j'ai fait équipe durant toutes ces années. Ses conseils et son support, ô combien précieux, émaillent l'ensemble du travail présenté. Longue vie aussi à nos équipées toulousaines, pyrénéennes, alsaciennes, ...

Merci aussi à Bauzil et Lemaire auxquels on doit des robots toujours frais et dispos, même lorsque l'inexorable pression des manipulations "très importantes et urgentes" se fait sentir. Merci fraternel à Maher toujours attentif aux copains et à leurs petits et grands problèmes.

Ce fut également avec un grand plaisir et une merveilleuse efficacité que j'ai eu l'occasion de travailler avec Jean-Paul Laumond, chercheur et voisin de quartier.

La dernière étape de ce travail m'a donné l'agréable opportunité de prendre contact, en leur qualité de rapporteurs, avec Bernard Espiau -Directeur de Recherche à l'INRIA-, Oussama Khatib -Professeur à l'Université de Stanford-, et Jean Louis Lacombe -Directeur adjoint de la recherche, MATRA-. Je les remercie vivement pour le soin avec lequel ils ont lu ce manuscrit et la qualité de leurs critiques ... en espérant d'autres occasions de collaborations. Merci également à Marc Courvoisier -Professeur à l'Université Paul Sabatier- d'avoir accepté de participer à ce jury de thèse.

Et je n'oublie pas toute la merveilleuse équipe RIA, son ambiance empreinte par chacun et, je le pense, appréciée par tous. Salut à Jackie. Salut aux générations passées, présentes et futures de thésards. Salut aux éternels thésards de mon cœur: Pichou, Paul, Simon, Philippe, Victor, Vianney. Salut à ceux qui m'ont fait découvrir la rochelle et ses animations extra-rochelaises: JJ, Fawzi, Bernard-le-motard, Hervé, Fifi, ... Salut à ceux avec qui j'ai tiré des bords inoubliables: Christophe, Patricia, Stéphane, Jean-Phi, Béatrice, Philippe. Salut à mes compagnons de bureau: Eric, Ralph, ... Salut à l'équipe haute en couleurs de la rochelle et d'ailleurs: Ricardo, Samer, Florent, Sep, Bertrand, Adelardo, ..., et à tant d'autres encore.

Merci enfin à ceux du "magasin", de la "doc", de la "cafet" : leur compétence et leur inébranlable bonne humeur se propagent à l'ensemble du labo.

Je reserve ma dernière pensée à mon très cher, très tendre et infiniment patient Laurent.

Table des matières

—

Table des figures

Table des matières

Introduction **1**

I Architectures de contrôle temps réel pour robots autonomes **3**

1 Exigences et choix conceptuels **5**
1.1 La problématique générale 5
1.2 Les différents niveaux d'autonomie 6
1.3 Délibération et réaction 8
1.4 Architecture du système robotique 9
1.5 Exemple d'architecture de contrôle 11
1.6 Conclusion 13

2 Etat de l'art **15**
2.1 Les approches temps réel 15
2.2 Les approches architecturales 24
2.3 Conclusion 27

II La couche fonctionnelle modulaire **31**

1 La couche fonctionnelle **33**
1.1 Propriétés et caractérisation 33
1.2 L'activité de la couche fonctionnelle: Les activités 36
1.3 Un réseau de modules 39
1.4 Le flux de contrôle: les requêtes 40
1.5 Le flux de données: les posters 45

2 Anatomie d'un module **49**
2.1 Définition d'un module 50
2.2 Contrôle d'une activité 50
2.3 Activité et fonctionnalité 53
2.4 Les interactions à l'intérieur d'un module 58
2.5 Structure logicielle d'un module 58

3 Description formelle et génération automatique **63**
3.1 Description formelle 63
3.2 Le générateur de modules G^{en}oM 70
3.3 Méthode de développement d'un module 72
3.4 Conclusion 76

III Intégrations et expérimentations **77**

1 Présentation de la couche fonctionnelle d'Hilare2 **79**
1.1 Description du robot *Hilare2* 80
1.2 La couche fonctionnelle d'*Hilare2* 82
1.3 Simulation 98
1.4 Interface utilisateur 99
1.5 Conclusion 100

2 Développement de fonctions pour le déplacement et la localisation **101**
2.1 Contrôle des mouvements de véhicule non-holonome 101
2.2 La localisation externe 110
2.3 Conclusion 115

3 EDEN: Une application en robotique d'extérieur **117**
3.1 Le robot ADAM 118
3.2 La problématique 118
3.3 Les solutions mises en œuvre 119
3.4 Conclusion 125

4 STRADA: Une application multi-robots **127**
4.1 La problématique générale 129
4.2 La couche fonctionnelle 134
4.3 Les interactions entre les modules et le superviseur 139
4.4 Les interfaces graphiques 140
4.5 Une simulation complète et réaliste 140
4.6 Une expérimentation à trois robots 141

Conclusion **147**

Annexes **149**

A Grammaire de G^{en}$_{o}$M **151**

B Génération d'un module par G^{en}$_{o}$M **153**

C Asservissement par retour d'état **158**

D Génération de trajectoires **161**

E Localisation externe **167**

Références bibliographiques **173**

Table des figures

1.1 L'autonomie d'un robot mobile: les niveaux de réaction et de décision. 7
1.2 Intégration de l'exécution et de la planification. 9
1.3 Un exemple d'architecture de contrôle complète. 11
1.4 Boucle d'interprétation de C-PRS 12

2.1 L'architecture purement réactive par "subsomption". 26
2.2 L'architecture centralisée TCA. 26
2.3 L'architecture hiérarchique NASREM. 27
2.4 Organisation "classique" des systèmes distribués temps réel. 29

1.1 Le rôle de la couche fonctionnelle 34
1.2 Arbre d'activités. 36
1.3 Exemple d'action réflexe. 37
1.4 Structuration en module et interactions inter-modules. 41
1.5 Couche fonctionnelle du robot Hilare2. 41
1.6 Une relation client/serveur. 41
1.7 Les deux types de requête. 42
1.8 Le flux de données: les posters. 45
1.9 Poster de contrôle et posters d'exécution. 46
1.10 Exemple de redirection de flux de données. 47

2.1 Structuration et fonctionnement d'un module. 50
2.2 Graphe de contrôle d'une activité. 52
2.3 Graphe d'état fonctionnel d'une activité. 54
2.4 Une instance particulière du graphe d'état. 55
2.5 Exemple de chronogramme d'activités 57
2.6 Interactions intra-modulaire 59
2.7 Structure logicielle d'un module. 59

3.1 Le générateur et le cycle de développement 71
3.2 Une interface de test. 74

1.1 Structure informatique générale. 81
1.2 Structure informatique embarquée. 82
1.3 Les interactions d'un module: notations. 83
1.4 Les modules de base de la couche fonctionnelle d'Hilare2. 83
1.5 Structure du module LOCO 85
1.6 Les principales interactions du module LOCO. 86
1.7 Les principales interactions du module US. 89
1.8 Visualisation des échos ultrason. 90
1.9 Les principales interactions du module TELE3D. 91
1.10 Les principales interactions du module PILO. 92
1.11 Chronogramme des activités pendant l'exécution d'une trajectoire. 93
1.12 Chronogramme des activités pendant l'exécution d'une trajectoire complexe. 93
1.13 Les principales interactions du module AVOID. 94
1.14 Exemple de flux de contrôle et de flux de données 95
1.15 Evitement d'un obstacle lors de l'exécution d'une trajectoire. 95

1.16 Les principales interactions du module LOCA2D. 96
1.17 Localisation extéroceptive. 96
1.18 Les principales interactions du module LOCEXT. 98
1.19 L'interface graphique de contrôle du robot. 99

2.1 Le modèle cinématique d'*Hilare2*. 102
2.2 Asservissement sur une trajectoire. 103
2.3 Enchaînement des clothoïdes et des anticlothoïdes. 107
2.4 Trajectoire composée de clothoïdes et d'anticlothoïdes, et profils de vitesses. 107
2.5 Lissage d'une ligne brisée: les quatre types de virages. 108
2.6 Lissage d'une ligne brisée orientée. 109
2.7 Localisation externe: le graphe des repères. 110
2.8 Exemple d'identification. 112
2.9 Synopsis de la procédure de localisation. 113
2.10 Suivi de trajectoire selon l'odométrie et selon les caméras externes. 114
2.11 Points extraits d'une image perçue par une caméra. 115

3.1 Le robot ADAM. 118
3.2 L'architecture de contrôle et le flux de données entre les modules. 120
3.3 Classification du terrain. 121
3.4 Une trajectoire 3D planifiée sur une carte d'élévation numérique. 122
3.5 La couche fonctionnelle d'Adam : les flux de données. 123
3.6 La boucle principale de navigation. 125
3.7 La procédure d'exécution de trajectoire 2D. 126

4.1 Le robot Commutor. 128
4.2 Le nouveau port de Rotterdam. 128
4.3 Un environnement et son modèle topologique. 130
4.4 Exemple de coordination à un carrefour. 132
4.5 Exemple de synchronisation sur trajectoires. 132
4.6 Architecture du superviseur du robot. 133
4.7 Modules, posters et flux de données de l'application MARTHA. 135
4.8 Intégration de la simulation MARTHA. 140
4.9 Un modèle d'environnement avec 10 robots simulés. 141
4.10 Les trois robots Hilare de l'expérimentation STRADA. 143
4.11 Le contexte expérimental. 143
4.12 La couche fonctionnelle des robots. 144
4.13 Le plan de la salle robotique et du couloir d'accès. 145
4.14 Visualisation graphique 3D des trois robots. 146
4.15 Visualisation de la même scène en 2D avec les synchronisations. 146

B.1 Exemple d'organisation pour le développement d'un module. 154

C.1 La cinématique d'*Hilare2*. 158

D.1 Une clothoïde. 162
D.2 Une anticlothoïde. 162
D.3 Enchaînement droite-cercle par une clothoïde. 165

E.1 Relations entre le robot et le motif après un déplacement. 169
E.2 La trajectoire selon la localisation externe et selon l'odométrie. 171
E.3 Les hyperboles obtenues expérimentalement. 171

Introduction

Un robot mobile autonome doit effectuer des tâches non répétitives dans un environnement imparfaitement connu et non-coopératif, voire hostile. Dans ce contexte, les missions attribuées au robot ne peuvent être définies que de façon abstraite et peu détaillée, et le robot doit être doté de moyens lui permettant d'interpréter la mission en fonction du contexte d'exécution, d'appréhender l'environnement et de réagir à des événements imprévus. La spécificité de l'architecture de contrôle d'un tel robot réside dans sa capacité à concilier *décision* et *réaction*. Afin de répondre à cette double exigence, l'architecture proposée comporte deux niveaux hiérarchiques. Le niveau *décisionnel* analyse la mission, la décompose en tâches exécutables, décide des actions à accomplir en fonction de ses connaissances sur l'environnement et sur l'état du robot, et contrôle le déroulement de ces actions. Le niveau *fonctionnel* (ou couche fonctionnelle) fournit au niveau décisionnel les moyens d'exécuter ces actions; il regroupe l'ensemble des fonctions opératoires qui vont permettre au robot de se déplacer, de percevoir et de modéliser son environnement, de calculer des trajectoires ou encore d'estimer sa position.

La couche fonctionnelle se trouve ainsi à la jonction entre le niveau décisionnel et le monde réel dans lequel évolue le robot. C'est grâce à elle que des informations interprétables par des processus de raisonnement de plus haut niveau sont extraites des données numériques issues des capteurs et que le robot maintient une interaction permanente avec son environnement: elle est le lieu des asservissements et des actions réflexes. Ces différentes actions doivent satisfaire de nombreuses contraintes relatives aux systèmes temps réel réactifs: elles doivent pouvoir être activées ou désactivées en des temps bornés, réagir à des situations critiques telles que l'apparition d'obstacles, s'exécuter de façon coordonnée, transférer efficacement des données et, en particulier dans le cas d'activités périodiques telles que les asservissements, respecter des temps de traitement.

Ainsi, la couche fonctionnelle se présente simultanément comme un contexte d'exécution temps réel, une bibliothèque de services, une image de l'activité et de l'état du robot, et une base d'informations relatives à l'environnement perçu par le robot.

Ces multiples aspects et la diversité des fonctions qui composent la couche fonctionnelle nous ont conduit à la structurer en *modules*. Les modules sont des entités informatiques qui intègrent des fonctions spécifiques et dont les moyens d'action sur le système robotique sont la production de données, la commande de capteurs ou d'actionneurs, ou encore le contrôle d'autres modules. Une tâche du robot est alors accomplie en connectant dynamiquement les modules.

La formalisation de la structure, du comportement et des fonctionnalités des modules a abouti à l'élaboration d'un langage de spécification dont l'objet est la déclaration complète de ces modules: les services offerts, les données manipulées et les fonctions de traitement associées. S'appuyant sur cette déclaration, un générateur automatique nommé G^{en}oM [1] produit les modules, et des fonctions de test, directement exécutables sur une machine cible

1. Generator of Modules. Par extension G^{en}oM est également le nom du système de conception et d'intégration de modules.

multi-processeurs UNIX ou VxWorks. G^{en}oM produit également des bibliothèques de fonctions d'interaction avec ces modules qui permettent de les intégrer à l'architecture de contrôle.

De nombreuses fonctionnalités ont ainsi pu être développées et intégrées aux architectures de contrôle de différents robots autonomes, acteurs de multiples expérimentations exposées dans ce mémoire. Le mémoire est organisé en trois parties:

▷ **La première partie** expose les problèmes que pose le contrôle d'un robot mobile autonome et les principales approches.

Nous présenterons dans un premier temps les principes et les choix conceptuels qui guident le développement d'une architecture de contrôle. Ils sont déduits de nombreuses études menées au LAAS et ont été affinés lors d'expérimentations variées et réalistes (chapitre I.1). Nous compléterons cette analyse par un panorama des différentes approches en nous focalisant plus particulièrement sur les travaux concernant la conception et l'intégration de systèmes informatiques temps réel. Nous citerons également les approches architecturales les plus représentatives, qui se sont traduites par ailleurs par de multiples variantes et à de nombreuses études comparatives (chapitre I.2).

▷ **La seconde partie,** le cœur du mémoire, est consacrée à la couche fonctionnelle, aux modules qui la composent et au générateur de modules.

Après une analyse générale des propriétés que doit satisfaire la couche fonctionnelle, nous proposerons une représentation de l'ensemble des actions qui s'y déroulent en termes *d'activités* et justifierons leur organisation en *modules*. Nous nous attacherons également à décrire les *requêtes* qui sont les vecteurs du *flux de contrôle* de la couche fonctionnelle et les *posters*, vecteurs du *flux de données* (chapitre II.1). Enfin seront détaillés la formalisation générique et la structure logicielle des modules (chapitre II.2), puis le langage de spécification et le générateur de modules G^{en}oM (chapitre II.3).

▷ **La troisième partie** présente les expérimentations qui mettent en œuvre notre système et qui démontrent son efficacité.

Nous décrirons dans un premier temps les huit modules de base qui composent la couche fonctionnelle du robot *Hilare2*. Nous suiverons à cette occasion un exemple de développement d'un module au moyen de G^{en}oM, ainsi que les interactions entre les modules durant leur exécution (chapitre III.1). Le chapitre suivant traitera des principaux algorithmes, mis en œuvre par trois modules particuliers que nous avons développé, qui ont trait à la production de trajectoires non-holonomes, au contrôle des déplacements du robot et à sa localisation par un système de caméras externes (chapitre III.2).

Enfin, les deux derniers chapitres présenteront deux expérimentations qui intègrent la couche fonctionnelle et un niveau décisionnel dans une architecture de contrôle complète qui procure son autonomie au robot. EDEN est une Expérimentation de Déplacement en Environnement Naturel, réalisée sur le robot ADAM dans le contexte de la robotique d'exploration planétaire. Elle consiste à réaliser la mission canonique "ALLER_A (but)" dans un environnement naturel, non structuré et complètement inconnu au départ (chapitre III.3). La seconde, initiée par le projet Esprit MARTHA de coordination d'une flotte de robots dans un environnement structuré de type réseau routier, a débouché sur une simulation réaliste faisant intervenir jusqu'à 15 robots, et surtout sur l'expérimentation STRADA au cours de laquelle trois robots de la famille *Hilare* se coordonnent de façon autonome dans le cadre d'un environnement d'intérieur (chapitre III.4).

Première partie

Architectures de contrôle temps réel pour robots autonomes

Chapitre 1

Exigences et choix conceptuels

Les travaux présentés dans ce mémoire concernent l'organisation, le développement et l'intégration de l'ensemble des fonctions opératoires dont doit disposer un robot mobile autonome afin d'appréhender l'environnement et d'agir selon la tâche qui lui a été assignée. Il s'agit ici des fonctions de perception, de commande et de modélisation. Cependant, la conception d'un robot mobile autonome, qui soit capable d'accomplir une variété de tâches dans un environnement mal connu et évolutif, nécessite également des capacités décisionnelles qui vont lui permettre d'interpréter la mission et de déterminer les actions adéquates selon le contexte.

Ainsi les fonctions opératoires, regroupées au sein d'une couche fonctionnelle, sont au service de niveaux décisionnels et doivent s'intégrer dans une structure comportant différentes entités organisées communicantes: l'architecture de contrôle du robot.

C'est pourquoi, dans ce premier chapitre, nous allons présenter la problématique générale du contrôle des robots mobiles autonomes, les concepts déduits de nombreuses études menées par l'équipe RIA [1] du LAAS et les principes architecturaux que l'on devra en particulier considérer lors de la conception de la couche fonctionnelle.

1.1 La problématique générale

Les domaines d'application effectifs ou potentiels des robots d'intervention ou de service sont nombreux: surveillance de sites industriels ou de lieux publiques, aide aux handicapés, transitique, maintenance de centrales nucléaires, de pipe-lines sous-marins ou de satellites, intervention après une catastrophe, exploration planétaire (Lune, Mars) ou sous-marine.

Dans ce cadre, le robot mobile est amené à accomplir des tâches avec un certain niveau d'autonomie dans un environnement qui n'a pas été conçu pour coopérer avec la machine. Contrairement à la robotique manufacturière, il ne s'agit pas de substituer l'homme par la machine mais de disposer d'un outil "intelligent" contrôlé par l'opérateur et qui lui donne les moyens d'intervenir dans des milieux hostiles ou de le soulager de travaux pénibles ou dangereux.

On conçoit que le niveau d'autonomie du robot est très dépendant du type d'application,

1. Robotique et Intelligence Artificielle.

et particulièrement des contraintes dues au contexte d'évolution, et des possibilités d'intervention de l'opérateur dans la description de la tâche et le contrôle de son exécution.

Ainsi, dans un environnement non structuré et/ou mal connu le système d'intervention découvre l'environnement et doit s'y adapter au fur et à mesure de la mission. A défaut de données complètes préalables concernant les conditions d'exécution, les objectifs de la mission ne peuvent être spécifiés qu'avec un certain niveau d'abstraction. Le système doit donc disposer de capacités pour percevoir et modéliser son environnement, pour analyser la situation et décider des actions pour satisfaire l'objectif fixé. L'appréhension de l'environnement est d'autant plus cruciale qu'il est sujet à des changements au cours du temps; il doit alors être l'objet d'une observation permanente par le système qui devra réagir face à cette évolution, au moins pour assurer sa propre sécurité. Le système doit donc tendre de façon cohérente vers l'objectif sous l'impulsion simultanée des données relatives à l'environnement, acquises au moyen de capteurs, et des commandes transmises par l'opérateur.

Le niveau d'intervention de l'opérateur est également dicté par les performances des médiums de communications avec le système robotique qui se caractérisent par le débit, les délais et les intermittences des transmissions. La télécommande ou la téléopération d'un système ne supportent pas ou peu de retard et requièrent généralement des débits importants. Par exemple dans le cadre d'opérations sur sites distants (Lune, Mars, fonds marins, mais aussi les bâtiments plus ou moins opaques à des transmissions hertziennes) les tâches ne peuvent être transmises que de loin en loin et le système devra disposer de capacités décisionnelles pour interpréter et dérouler correctement sa mission. Outre ces limitations techniques, l'autonomie peut être requise pour soulager l'homme de contrôles complexes, et notamment pour des raisons de sécurité (aide à la conduite, coordination de nombreux robots en transitique, aide aux handicapés).

Il ressort de ce premier aperçu que le robot mobile autonome est une machine programmable à un certain niveau d'abstraction, dont le niveau d'autonomie dépend des capacités de réaction et de décision.

1.2 Les différents niveaux d'autonomie

L'autonomie d'une machine peut être définie comme la capacité d'exécuter une action ou une séquence d'actions sans intervention d'un opérateur et quelles que soient les péripéties qui accompagnent l'évolution du système. Il y a naturellement une limite à de telles variations au delà de laquelle le système s'avère impuissant à satisfaire le but fixé.

Le degré d'autonomie va se caractériser par la variété des situations que le système peut appréhender, et va se traduire par l'expression plus ou moins précise ou directive de la nature des tâches qui lui sont assignées, c'est à dire par leur sémantique.

Regardons de plus près les différents niveaux d'autonomie que peut posséder un robot (figure 1.1 page ci-contre). Nous en déduirons les facultés dont doit être doté un robot mobile autonome et les implications quant à la conception d'un tel système.

1. **Commande par retour d'état:** l'asservissement du système selon une consigne paramétrable et son état interne, mesuré au moyen de capteurs proprioceptifs, constitue le premier niveau de contrôle d'un robot. Ce niveau d'autonomie est celui dont disposent les robots de première génération qui peuvent ainsi être programmés afin de réitérer

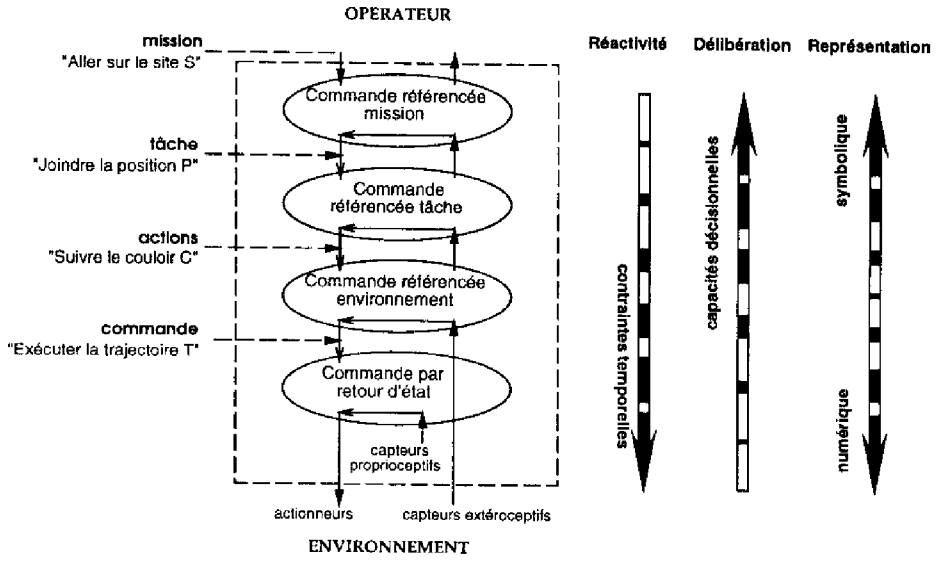


FIG. 1.1 – *L'autonomie d'un robot mobile: les niveaux de réaction et de décision.*

une opération donnée dans un environnement invariant. Une action typique consiste à exécuter de façon répétitive une trajectoire $\{(x_i, y_i)_{i\in[1..N]}\}$ dont les paramètres sont entièrement définis à la programmation.

2. **Commande référencée sur l'environnement:** l'adaptation du robot à son environnement passe par des boucles de perception/action qui permettent de s'asservir relativement à l'environnement et d'installer des actions réflexes. Cette capacité caractérise les robots de seconde génération. On peut ainsi surveiller la présence d'obstacles, saisir une pièce dont la position est incertaine au moyen de données perceptuelles, ou suivre une trace au sol (filoguidage).
3. **Commande référencée tâches:** afin d'élargir le domaine d'applicabilité de la tâche, le robot ne peut être uniquement guidé par des stimuli mais ses actions doivent être sélectionnées et instanciées en fonction de la situation et bien sûr de la finalité tâche. Il s'agit d'un premier niveau de délibération qui, selon des règles prédéfinies, attribue au robot une certaine autonomie relativement à la sélection des actions adéquates et à leur paramétrisation en fonction de données contextuelles acquises et interprétées par le robot. Par exemple, la tâche "Joindre la position P" peut se traduire par un recalage préalable si la position du robot est trop incertaine, puis par le calcul d'une trajectoire qui sera exécutée en évitant les obstacles locaux.
4. **Planification des tâches:** dans des environnements non connus a priori ou sujets à des évolutions importantes, ou bien lors de missions longues pouvant comporter de nombreuses incertitudes (missions planétaires, interventions après une catastrophe, coordinations avec d'autres systèmes indépendants), l'énoncé précis des tâches n'est pas toujours possible. La mission se présente alors sous la forme d'un objectif à atteindre et le robot doit être doté de capacités de raisonnement pour décider des tâches à exécuter. Par exemple, dans le cadre d'une mission d'exploration planétaire, la mission "Aller sur le site A" nécessite des stratégies de déplacement et de perception afin de déterminer

le chemin le mieux adapté en intégrant différents critères (durée, coût énergétique, sécurité), ou encore dans le domaine de la transitique la mission "Décharger le bateau B" peut impliquer des coordinations avec les robots qui partagent des ressources communes (le réseau routier, la station de déchargement). Le ou les planificateurs intégrés sur le robot sont très liés au type de mission: planification avec contraintes temporelles, coordination avec d'autres robots autonomes, stratégies d'exploration, *etc.*

Un robot est dit autonome s'il est capable d'exécuter une classe de tâches dans un environnement qui n'a pas été conçu pour lui, c'est à dire s'il dispose de capacités pour appréhender l'environnement, réagir lors d'évolutions de celui-ci et raisonner sur la situation: le robot mobile autonome doit intégrer au minimum les aptitudes relatives aux trois premiers niveaux d'autonomie précédemment énoncés. La difficulté que pose la conception d'un système robotique est de concilier deux facultés a priori antinomiques: la *délibération*, dont la durée dépend de la complexité du problème à résoudre, et la *réaction*, qui doit être en adéquation avec la dynamique de l'environnement de la tâche.

1.3 Délibération et réaction

Il ressort que le robot doit d'une part permettre de mettre en œuvre des actions réflexes et des boucles de perception/action très proches des capteurs et des actionneurs, et d'autre part disposer de capacités décisionnelles pour interpréter des missions "abstraites" et décider des actions futures.

Cependant, cette apparente dichotomie entre décision et réaction ne reflète pas l'ensemble des contraintes que doit satisfaire le système robotique. L'évolutivité de l'environnement, ou tout au moins l'évolution de la perception de cet environnement au fur et à mesure de l'avancée de la mission et des déplacements du robot, rend la réactivité nécessaire à *tous les niveaux de décision* et non pas uniquement sous la forme d'actions réflexes.

La délibération doit pouvoir intégrer des événements asynchrones, qui marquent des changements qualitatifs du contexte d'exécution de la tâche, et réagir en temps borné. Pour décider des prochaines actions, elle doit être capable d'évaluer l'état courant du robot et de l'environnement et d'anticiper sur leurs états futurs. La délibération apparaît donc simultanément comme un processus de planification en ligne dirigé par le but afin d'anticiper sur l'évolution de l'environnement, et comme un système de prise de décision temps réel afin de répondre aux événements de façon opportune.

La délibération repose également sur des mécanismes de modélisation et de mémorisation ([Chatila 93a]). En effet, les données brutes issues des capteurs ne sont pas exploitables directement par les processus décisionnels et, en raison de leur occupation mémoire, ne peuvent être stockées telles quelles. Des représentations adéquates devront être élaborées en fonction de la tâche à accomplir. La diversité de ces tâches se traduit par une diversité des représentations et pose les problèmes du transfert des données entre les processus, de la construction incrémentale des modèles, de la fusion, de la cohérence et de la disponibilité en temps borné des informations.

Si l'on considère par exemple la navigation en terrain accidenté, le processus d'évitement d'obstacle doit disposer d'informations temps réel, fugaces et peu affinées. Le planificateur de trajectoire "3D" utilise un modèle numérique du terrain qui ne peut être conservé que tem-

porairement étant donnée sa taille. Par contre, le processus de localisation doit maintenir une carte cohérente d'amers extraits d'une succession d'images de profondeur. Enfin, la stratégie de déplacement est élaborée à partir d'un modèle topologique construit incrémentalement à partir de la classification du terrain selon sa praticabilité (plat, accidenté, obstacles).

L'organisation des capacités du robot, c'est à dire la façon dont interagissent la représentation, le raisonnement et les réactions (figure 1.1 page 7) est un problème central qui relève de l'*architecture du système.*

1.4 Architecture du système robotique

Afin de concilier décision et réaction, et en s'appuyant sur des expérimentations, des concepts pour une architecture générique ont été développés au LAAS [Giralt 93, Chatila 93a, Alami 93]. Cette architecture distingue deux grands niveaux:

- *un niveau décisionnel* organisé en couches qui inclut les capacités de planification et de contrôle du déroulement de la tâche,
- *un niveau fonctionnel* qui intègre les fonctions de contrôle des capteurs et des actionneurs, d'asservissement, de surveillances et de calcul.

1.4.1 Le niveau décisionnel

Le niveau décisionnel doit satisfaire deux fonctions dont les temps de traitement ne sont pas compatibles: le contrôle d'exécution des tâches et leur planification. En effet, la planification requiert généralement un temps de calcul plus long que la dynamique imposée par le système contrôlé. Ainsi, ce niveau est lui-même structuré en deux parties. Un *superviseur* qui interagit avec le niveau inférieur, contrôle le déroulement des actions et réagit aux événements avec un temps de réaction borné et qui transmet les opérations de planification à un *planificateur* qui produit les séquences d'actions nécessaires (figure 1.2).

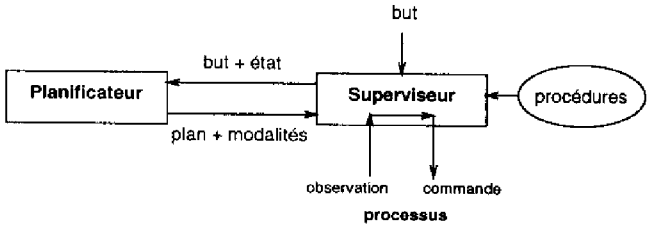


FIG. 1.2 – *Intégration de l'exécution et de la planification.*

▷ **Le planificateur** reçoit une description de l'état du monde et un objectif; il produit en retour un plan composé de séquences d'actions et de modalités d'exécution. Les modalités d'exécution décrivent les contraintes à respecter durant le déroulement du plan, et les situations à surveiller associées à des réactions appropriées en cas d'urgence. Un plan robuste doit en effet envisager les situations non-nominales, et en particulier les situations critiques, et prévoir des réactions adaptées. Les réactions sont des actions réflexes, des variantes prévues du plan ou des requêtes de replanification.

▷ **Le superviseur** interagit avec les autres couches et le planificateur. Les autres couches sont vues comme un ensemble de processus qui échangent des signaux et des données avec le superviseur. Ces processus, sous le contrôle du superviseur, correspondent aux actions du

robot et reflètent l'évolution des relations robot/environnement. Le superviseur doit satisfaire leurs caractéristiques propres. Par exemple, un processus qui correspond au mouvement du robot comporte une certaine inertie et ne peut être interrompu de façon instantanée par le superviseur. Le superviseur peut requérir un arrêt à n'importe quel instant de l'exécution mais le processus transitera par une série d'étapes avant de se terminer effectivement. Ces processus peuvent être représentés par des automates à états finis. L'activité du superviseur consiste à surveiller l'exécution du plan en procédant à des détections et évaluations de situations, et en prenant les décisions appropriées en temps réel, c'est à dire dans des limites de temps compatibles avec la dynamique des processus contrôlés. Le superviseur ferme la boucle de perception/décision/action selon les consignes du plan et doit être intégré au moyen d'algorithmes délibératifs qui garantissent une réponse en temps borné. Les opérations qui ne peuvent satisfaire cette contrainte sont déléguées au planificateur sur requête du superviseur. Afin de réagir au plus tôt, ce dernier utilise un ensemble de procédures prédéfinies qui sont sélectionnées et instanciées selon la situation et le plan courant.

Le niveau décisionnel peut lui-même être constitué de plusieurs couches selon le détail des représentations manipulées et le type de planification à accomplir. Une mission peut par exemple nécessiter un planificateur temporel qui organisera les différentes tâches en estimant leurs durées et le temps imparti, un planificateur de navigation qui décidera de la tactique à suivre pour atteindre un objectif, ou encore un planificateur chargé de coordonner les plans avec ceux d'autres robots mobiles autonomes, *etc.*

1.4.2 Le niveau fonctionnel

L'ensemble des actions du robot sont exécutées par le niveau fonctionnel qui intègre les commandes des capteurs et des actionneurs, les asservissements, les surveillances qui peuvent être associées à des actions réflexes et des procédures de calcul (planificateur de chemins, segmentation d'images, ...). La diversité des fonctions invoquées et contrôlées par le niveau supérieur exige une structuration de la couche fonctionnelle. Les fonctions sont regroupées en *modules* qui sont des entités informatiques hébergeant des traitements spécifiques mis en œuvre sur *requêtes*. Un module peut accéder à des données produites par d'autres modules, agir sur des dispositifs du robot et/ou sur d'autres modules, et exporter des données résultant des traitements. L'organisation des modules n'est pas figée et leurs interactions dépendent de la tâche à exécuter. Cette propriété importante permet d'obtenir un comportement flexible, programmable, et non systématique [Chatila 90, Fleury 94].

Outre cet aspect architectural, la couche fonctionnelle étant le contexte d'exécution simultané d'actions qui doivent réagir et s'exécuter en des temps bornés (asservissements, surveillances, actions réflexes), elle doit également satisfaire les contraintes inhérentes aux systèmes informatiques temps réel distribués. Elle doit offrir en particulier des mécanismes d'activation/désactivation, de synchronisation et de communication efficaces [Ferraz De Camargo 91, Perebaskine 92].

La couche fonctionnelle étant l'objet principal de cette thèse, ces différents aspects seront plus amplement développés par la suite.

1.5 Exemple d'architecture de contrôle

L'architecture très générale présentée sur la figure 1.3 s'applique aux robots d'intervention qui opèrent dans des environnements mal connus, découverts progressivement au moyen des capteurs embarqués. L'opérateur et le système robotique ne disposent initialement que d'un modèle du monde décrit à un niveau symbolique, et la mission ne peut être exprimée qu'en termes de buts associés à des contraintes particulières (priorité des buts, contraintes en temps, fenêtres de validité, ...).

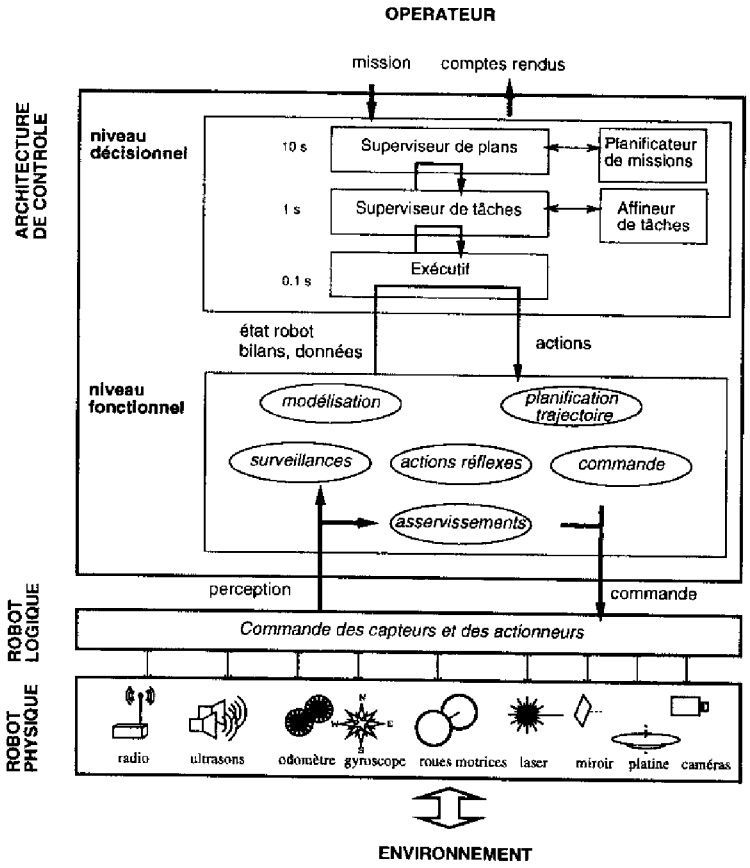


FIG. 1.3 – *Un exemple d'architecture de contrôle complète.*

L'architecture proposée est organisée en trois niveaux représentant deux niveaux décisionnels au-dessus du niveau fonctionnel. Les deux niveaux supérieurs sont construits selon le paradigme superviseur/planificateur. Le plus haut niveau utilise un planificateur temporel. Le second niveau reçoit les tâches et les transforme en procédures composées d'actions élémentaires, et supervise l'exécution de ces procédures tout en étant réactif aux événements asynchrones. La planification à ce niveau est un *affinement* qui utilise des connaissances spécifiques sur la tâche ou le domaine.

En ce qui concerne le niveau de planification de mission, il a été développé un planificateur temporel nommé IxTeT (Indexed Time Table) [Ghallab 88, Chatila 92, Laruelle 94]. Il permet de raisonner sur des relations temporelles numériques ou symboliques entre des instants. Un plan produit par IxTeT est un ensemble de tâches partiellement ordonnées associées à des contraintes temporelles telles que les bornes minimale et maximale sur les durées d'exécution

escomptées, et des synchronisations sur des événements externes attendus ou sur des dates absolues.

La supervision et de l'affinement de tâches ont été intégrés au moyen de PRS (Procedural Reasoning System) [Georgeff 87] qui fournit un contexte bien adapté pour implanter les interactions entre la délibération et la réaction. C-PRS (la version de PRS que nous utilisons, voir [Ingrand 92b]) propose des outils et des mécanismes pour représenter et exécuter des plans sous la forme de séquences conditionnelles d'actions invoquées à l'occurrence de buts ou de situations particulières. Les principaux composants de PRS sont: une base de données qui contient des faits représentant l'état du système et qui est mise à jour automatiquement à l'occurrence d'événements; une bibliothèque de procédures, ou scripts (nommés KA en PRS), qui décrivent des séquences particulières d'actions ou de surveillances qui seront invoquées pour satisfaire des buts donnés ou pour réagir à certaines situations; un graphe dynamique d'intentions (les tâches) en cours d'exécution. Les intentions sont des structures dynamiques qui exécutent les procédures sélectionnées selon les buts postés ou les faits présents dans la base de données. Ainsi les mécanismes d'inférence utilisés dans C-PRS sont capables de réagir à de nouveaux événements tout en poursuivant l'exécution des procédures déjà actives. La figure 1.4 schématise l'interpréteur de C-PRS.

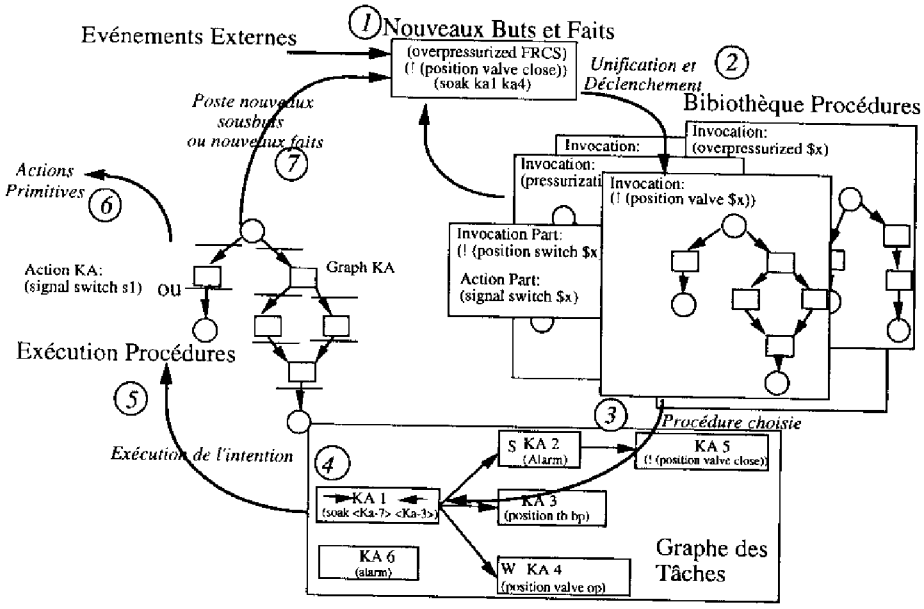


FIG. 1.4 – *Boucle d'interprétation de C-PRS*

L'exécutif, quant à lui, constitue un niveau purement réactif sans capacité de planification. Il contrôle l'exécution des actions selon des schémas prédéfinis et sélectionnés selon le contexte. Il est implanté au moyen du système à base de règles Kheops (voir [Philippe 89] et §2.1.1.1) qui permet de compiler hors-ligne un ensemble de règles produisant un programme qui consiste en une recherche en temps borné dans un arbre de décision.

1.6 Conclusion

La couche fonctionnelle, qui est l'objet de ce mémoire, a des contraintes propres relatives à l'intégration de processus temps réel mais elle doit également, comme on vient de le voir, satisfaire les exigences relatives à son intégration dans l'architecture globale du robot. Cette couche fonctionnelle doit être *programmable* par le niveau supérieur qui décide des activités à mettre en œuvre, à coordonner ou à interrompre. Elle doit être *réactive* non seulement vis à vis des événements issus des interactions robot/environnement mais également vis à vis des requêtes qui proviennent des couches de contrôle. Elle doit être *observable*, car c'est de cette observation que les niveaux de contrôle d'exécution et décisionnel décideront des actions immédiates et élaboreront les stratégies futures. Enfin, elle doit être *robuste* car elle est le dernier niveau de contrôle du robot physique et en assure la sécurité, en particulier par le biais d'actions réflexes.

Après une partie consacrée à la couche fonctionnelle, nous présenterons deux réalisations complètes d'architecture de contrôle qui respectent les choix et principes architecturaux présentés. Le première expérimentation, dans le cadre des projets VAP [2] puis IARES [3], concerne l'exploration en milieu naturel et planétaire: le robot ADAM reçoit d'une station distante la tâche canonique de navigation "Aller à (but)". La seconde application, dans le cadre du projet MARTHA [4] consiste à coordonner une flotte de robots. En raison de la dynamique de l'environnement et l'incertitude quant à la durée des actions, la station centrale ne peut élaborer de plan détaillé. Les robots doivent être dotés de tous les moyens nécessaires pour planifier, naviguer et se coordonner dans un contexte dynamique multi-robots. Cette application a donné lieu à une expérimentation réelle à trois robots.

2. Les partenaires du projet VAP (Véhicule Automatique Planétaire) sont le CNES (Centre National d'Etudes Spatiales) et le consortium RISP (Robots d'Intervention sur Site Planétaire) auquel appartient le LAAS.

3. Le projet Euréka IARES (Illustrateur Autonome de Robotique mobile pour l'Exploration Spatiale) a été entrepris par le CNES à la suite du projet VAP.

4. MARTHA (Mobile Autonomous Robots for Transportation and Handling Application) est un projet Esprit.

Chapitre 2

Etat de l'art

Depuis maintenant plus de deux décennies on cherche à concevoir des robots mobiles capables de se mouvoir et d'accomplir des tâches "intelligentes" dans des environnements non adaptés à la machine.

La planification classique en IA est longtemps restée la principale composante dans l'étude des robots intelligents. Le système PLANEX est le premier à avoir fait le lien entre un planificateur, STRIPS, et l'exécution sur un robot mobile, SHAKEY [Fikes 72]. Cependant, le concept qui a marqué une avancée importante dans le contrôle de robot, et qui a permis de quitter le "monde des cubes", est la *réactivité*. La prise en considération de l'évolutivité du monde, des interactions entre le robot et son environnement, et la nécessité de répondre en temps borné, ont ouvert de nouvelles classes de problèmes et ont conduit au développement d'architectures de contrôle qui restent encore aujourd'hui une des préoccupations principales dans la conception de robots mobiles autonomes.

D'autre part, les progrès considérables dans les domaines de l'électronique, des microprocesseurs, des capteurs et des actionneurs et de la commande, autorisent de meilleures performances et la parallélisation des calculs et des actions. L'algorithmique temps réel et les techniques de conception et d'intégration de systèmes temps réel viennent enrichir l'architecture logicielle des robots.

Ce chapitre est consacré à la présentation de ces deux grands axes de recherche:

- la spécification et l'intégration de systèmes multi-tâches temps réel complexes et,
- la conception d'architectures de contrôle décisionnelles réactives.

Nous nous focaliserons plus particulièrement sur le premier point et ses implications en robotique mobile autonome, le second point ayant été largement développé dans la littérature relative à ces robots et ayant conduit à de nombreuses analyses comparatives. Après une présentation de diverses approches, nous en proposerons quelques critiques et tirerons des conclusions dont découle notre approche.

2.1 Les approches temps réel

Comme cela a été évoqué au premier chapitre, l'architecture de contrôle d'un robot mobile autonome doit avoir des capacités de réactivité que ce soit au niveau des asservissements par retour d'état, des commandes référencées sur l'environnement, de la coordination des actions avec prise en compte en temps réel de l'évolution permanente du système ou encore de la

planification de stratégies qui dépendent de l'objectif mais aussi de l'état courant.

A chacun de ces niveaux, un système de contrôle doit maintenir une relation continuelle avec un procédé (axe de robot, ensemble d'actions du robot, *etc.*) qui progresse indépendamment de lui: il s'agit typiquement de *systèmes temps réel réactifs*. Ces systèmes doivent réagir, avec un temps de réponse imposé par le procédé contrôlé, aux signaux émis par ce procédé (ou aux mesures relevées) alors que les instants d'occurrence de ces événements sont a priori asynchrones par rapport au fonctionnement du système de contrôle.

Les systèmes réactifs posent de nombreux problèmes tels que la spécification de systèmes concurrents, la maîtrise de l'algorithmique parallèle (synchronisations, communications, ...), l'expression et la satisfaction de contraintes temporelles (durée, temps de réponse), de performance, de sûreté de fonctionnement (compliquées par le parallélisme des actions, la distribution matérielle et l'évolution indépendante du processus contrôlé), l'ordonnancement des tâches, leur répartition sur différents processeurs (imposée par une architecture matérielle distribuée ou par des contraintes de promptitude qui ne peuvent être satisfaites que par des exécutions en parallèle), ... Ces différents aspects, qui ne peuvent être approfondis ici, sont clairement exposés dans le rapport établi par le Groupe de Réflexion Temps Réel du CNRS [Elloy 88].

Il n'existe pas actuellement de méthode générale de spécification, de programmation et d'intégration de systèmes temps réel. Différents travaux ont cependant permis de mieux préciser certaines entités manipulables (événements, actions, requêtes, ...) et de développer des outils ou des méthodes telles que:

- des langages de programmation temps réel,
- des méthodes de spécification et de formalisation de systèmes,
- des méthodes de conception et d'intégration de systèmes structurés.

Nous présentons maintenant ces solutions, qui s'adressent à différentes classes de problèmes, en essayant de déterminer la portée de leur utilisation et leur applicabilité dans le cadre qui nous intéresse. On trouvera également une analyse intéressante dans [Dzierzgowski 90].

2.1.1 Les langages temps réel

Les langages temps réel cherchent à répondre au double objectif suivant:

- proposer un style de programmation mieux adapté à l'écriture de systèmes temps réel, en manipulant des notions d'événement et/ou de changement d'état qui, associées à des opérateurs pour exprimer le parallélisme, la synchronisation, *etc.*, permettent de décrire le comportement du système,
- valider le système programmé en s'appuyant sur un formalise mathématique rigoureux qui se traduit généralement à la compilation par un automate à états finis sur lequel des preuves de comportement pourront être établies.

Dans ce but, les domaines d'applicabilité doivent être très clairement définis car des hypothèses restrictives ont dû être avancées afin d'adapter les systèmes réactifs temps réel au formalisme mathématique:

- Si une action peut être considérée comme une opération élémentaire, alors satisfaire les échéances d'un ensemble d'actions revient à ordonner leur exécution. Si de plus les durées d'exécution des actions peuvent être considérées comme instantanées (vis-à-vis des dynamiques du procédé) alors l'expression du parallélisme des actions, de leurs

relations de précédence et de synchronisation, des conditions événementielles de leurs exécutions peut être traduite sous la forme d'un automate duquel tout parallélisme a disparu: il s'agit de l'approche *synchrone*.

- Si par contre ces hypothèses ne sont pas satisfaites parce que des opérations sont liées à la dynamique du processus contrôlé, ou parce que ce processus présente une variété de dynamiques (différents temps de réponse), alors l'approche *asynchrone* s'impose. Les actions pourront alors être préemptées et leur conditions de reprise d'exécution devront être définies.

2.1.1.1 Les langages synchrones

Dans le modèle synchrone les sorties sont synchrones avec les entrées, en d'autres termes les actions et l'émission/réception d'événements sont instantanées. La séquence des événements peut être assimilée à un temps logique. Cette hypothèse à première vue peu réaliste est cependant applicable à divers problèmes:

- les systèmes non temps réel (séquencement d'actions), et les systèmes temps réel lents pour lesquels il n'y a pas de problème de vitesse de calcul pour assurer la commande (contrôle d'ascenseurs, d'installations industrielles, d'interfaces souris/clavier).
- les systèmes dont les transitions sont cadencées par une horloge: programmation de circuits intégrés et traitement du signal (filtres analogiques substitués par des calculs numériques).
- les systèmes mettant en œuvre des traitements périodiques de durée constante (asservissements, surveillances).
- les systèmes complexes dont les multiples états et transitions possibles nécessitent une validation du programme et une gestion "optimale" de la combinatoire pour réagir en temps borné.

Ce dernier cas, très général, peut impliquer des adaptations pour satisfaire les hypothèses de synchronisme, soit en décomposant les actions qui ont une durée en des actions élémentaires, (ce qui risque de complexifier encore le système - *e.g.* décomposition d'un mouvement du robot en des mouvements élémentaires), soit en "délocalisant" le contrôle de ces actions et en ne considérant au niveau du système de contrôle principal que les transitions marquant les étapes clefs de ces actions (requête d'activation/d'arrêt, début/fin d'exécution).

Les différents langages élaborés, qui du point de vue de la logique sont équivalents, se distinguent essentiellement par le mode d'expression du problème, et de ce fait vont s'adresser à différentes classes d'applications.

▷ **Les langages à flots de données** SIGNAL [Guernic 91] et LUSTRE [Halbwachs 91] entrent typiquement dans cette catégorie. Ils sont une généralisation des modèles dynamiques utilisés pour le contrôle ou le calcul de signaux numériques. Le programme exprime les relations et les contraintes entre les signaux, c'est à dire entre les horloges[1] et les valeurs des différents signaux impliqués dans le système dynamique: ces systèmes sont également dénommés *Systèmes Récurrents à Horloges Multiples*[2]. Le formalisme de LUSTRE est proche des logiques temporelles, alors que SIGNAL a une approche plus géométrique en décrivant un réseau des opérateurs activés par les signaux.

1. Les signaux simultanés ont la même horloge.
2. Multiple-Clocked Recurrent Systems.

Ces langages sont plus particulièrement adaptés aux problèmes où les aspects liés aux flots de données sont prépondérants: traitements du signal (radar, reconnaissance vocale), circuits numériques (montres digitales, contrôleur d'aiguillages).

Un environnement graphique interactif de développement pour applications temps réel de traitement du signal et de contrôle de processus a également été développé: SYNDEX permet de concevoir et d'*implanter* des algorithmes sur des machines multi-processeurs en appliquant la méthodologie dite A^3 (Adéquation Algorithme/Architecture) [Lavarenne 94]. SYNDEX peut analyser directement des programmes SIGNAL. Ce système a permis la conception et l'intégration d'applications telles que des algorithmes de traitement d'images sur une architecture multi-DSP [Milan 93] ou des calculs massivement parallèles.

Le langage REX [Kaelbling 88], basé sur les mêmes concepts, construit un automate à partir d'un langage proche de COMMON LISP. Il a permis notamment d'élaborer un système de contrôle pour le robot Flakey du SRI (contrôle de mouvements avec détection d'obstacles et passage de portes sur données ultrasons). L'objectif est de garantir une borne maximale sur le temps de réponse du système en poussant la logique des systèmes synchrones jusqu'au bout: chaque action correspond à une instruction primitive et, afin de garantir un "tic" constant, l'ensemble des opérateurs de l'automate est parcouru à chaque période (y compris les branches inutilisées): la durée du tic est la durée d'exécution de la branche la plus longue.

▷ **Les langages basés sur les états (ou langages procéduraux).** ESTEREL [Berry 87] est un langage impératif qui décrit les actions à effectuer sous l'impulsion d'événements. Un programme ESTEREL exprime les différents états du système et distingue deux types d'entrée: les *signaux* qui sont des *tops* transitoires éventuellement associés à une valeur rémanente, et les *capteurs* que le système peut consulter quand il le souhaite. De même que pour les précédents langages synchrones présentés, la compilation d'un programme ESTEREL produit un automate à états finis. Ce langage, basé sur une description des états, s'applique plus particulièrement aux systèmes dont les aspects de contrôle sont prépondérants. Un type d'application fréquemment cité est le contrôle d'actions robotique.

Cependant, ce type d'applications s'adapte moins naturellement aux langages synchrones que ceux précédemment cités; des passerelles entre les processus asynchrones et le système de contrôle sont nécessaires: la primitive *exec* a été ajoutée au langage pour pouvoir considérer des actions qui ont une durée supérieure au temps minimal entre deux occurrences d'événements. Sa fonction est de démarrer des *tâches* externes dont le seul lien avec le programme ESTEREL est le début et la fin de l'exécution (pour pouvoir conserver l'hypothèse synchrone). L'interaction entre les événements asynchrones (événements externes, terminaison des tâches) et l'automate synchrone est assurée par une *machine d'exécution* qui enregistre les événements asynchrones et décide des instants d'activation de l'automate.

KHEOPS [Philippe 89] propose une approche très différente: le système est spécifié par un ensemble de règles (SI ... ALORS ...) qui présente le double avantage de ne pas avoir à déclarer explicitement tout l'automate et de pouvoir ajouter incrémentalement de nouvelles règles au fur et à mesure de l'évolution du système (adjonction de nouvelles fonctionnalités) sans remettre en cause les règles précédemment établies. Un aspect important de ce mode d'expression dans le cadre de systèmes complexes est de pouvoir définir explicitement des règles de *sécurité* générales (réactions face à des situations critiques) et d'expliciter les ressources critiques (détection de *conflits* sur des ressources telles que les actionneurs).

Les règles propositionnelles sont transformées en un réseau décisionnel *optimisé* au comportement déterministe. L'analyse effectuée au cours de la compilation (ou lors de l'adjonction de nouvelles règles dans un mode interactif) permet de lever toutes les inconsistances et les incomplétudes. Ce langage est (volontairement) plus limité que ceux précédemment présentés: il ne permet pas par exemple d'effectuer de calculs flottants. Son objectif est de modéliser le plus efficacement possible le comportement d'un système et de garantir l'obtention d'un et un seul résultat pour chaque entrée en un temps borné et optimal: l'automate étant optimisé, sa profondeur caractérise formellement le temps de réaction maximal. L'automate est activé par un système externe à l'occurrence de chaque événement (relatif à un ordre ou issu du processus contrôlé).

Ce langage est clairement dédié à la spécification et à l'intégration du contrôleur d'exécution centralisé de systèmes réactifs complexes: contrôle de turbines à gaz, de l'ensemble des actions d'un robot [Medeiros 93], ...

2.1.1.2 Les langages asynchrones

Tous les systèmes réactifs ne rentrent pas aisément dans le cadre de l'hypothèse synchrone, en particulier lorsqu'ils font intervenir des actions qui ont une durée et qui peuvent émettre ou recevoir des événements durant leur exécution (déplacement d'un robot, action d'orientation d'une caméra, surveillances passives, calculs complexes ...).

Si l'on excepte les langages généraux tels que ADA ou OCCAM, qui comportent quelques opérateurs permettant d'exprimer le parallélisme et la synchronisation [Padiou 90], mais sans formalisation du comportement, il existe très peu de langages réactifs asynchrones.

Le langage ELECTRE conçu au LAN [Roux 92] s'inspire en quelque sorte des langages synchrones dans la mesure où il permet également de produire un automate à états finis pour dérouler l'exécution et valider un certain nombre de propriétés (des études communes ont d'ailleurs été menées). Il définit deux types d'objets: les *modules* qui sont des portions de code séquentiel sans point de synchronisation bloquant (*i.e.* sans attente d'événement), et les *événements* qui cadencent l'évolution du système. La durée d'exécution d'un module est finie mais non nulle, et il peut durant cette exécution émettre des événements ou être interrompu par certains événements. La terminaison d'un module produit un événement implicite. Par hypothèse d'asynchronisme, les événements sont observés et traités immédiatement par les modules. Le langage permet d'exprimer qu'un module n'est pas préemptif, ainsi, si tous les modules sont déclarés ininterruptibles on se ramène strictement à un système synchrone.

Cette approche permet de définir des actions longues (*e.g.* le déplacement d'un robot) par un module et d'exprimer les possibilités d'interruption et de reprise d'un module. Les événements se caractérisent par différentes propriétés (fugaces, incrémentals, mémorisés).

Les opérateurs de base du langage sont le séquencement, la parallélisation et la réitération des modules, et sont enrichis par l'expression de la préemption et de l'activation conditionnelles des modules sous l'impulsion d'événements.

L'expression ELECTRE est déterministe en ce sens que la même séquence d'occurrence d'événements conduit toujours à un comportement identique du programme de contrôle. La compilation d'un programme ELECTRE produit un automate d'états finis qui permet de valider certains aspects tels que l'absence d'inter-bloquage ou d'états inatteignables. L'automate permet à un *exécutif spécialisé* de cadencer les processus selon les événements émis par le

procédé physique et en déduit les opérations à effectuer sur les modules.

L'asynchronisme augmente le pouvoir d'expression du langage, la contre-partie étant la difficulté de vérification des échéances temporelles. Les auteurs présentent deux exemples: l'expression du problème des lecteurs/écrivains et la gestion d'une souris.

2.1.2 Spécification et formalisation des systèmes réactifs distribués

Les systèmes réactifs distribués sont des systèmes qui, de par la combinatoire et la dynamique des interactions (ou transitions, selon la représentation) entre leurs multiples composants (ou états) sont difficiles à appréhender. Pour faire face, de nombreuses méthodes de spécification et outils de validation, pour l'essentiel graphiques, ont été proposés. On peut les classer en deux catégories ([Marty 94]):

- Les méthodes de modélisation focalisées sur le contrôle qui offrent une description événementielle du système.
- Les méthodes issues de l'analyse structurée qui proposent une décomposition fonctionnelle du système.

▷ **Modèles événementiels:** Ces approches offrent des représentations visuelles claires du séquencement des actions en se basant sur un graphe d'états. L'apparition d'événements provoque instantanément la (ou les) transition(s) valide(s): il s'agit typiquement de modèles synchrones. Parmi ces approches, les réseaux de PETRI sont les plus largement répandus et peut être les mieux formalisés. Leur syntaxe et leur sémantique rigoureuses font apparaître le flux de contrôle et permettent de démontrer un certain nombre de propriétés. Ils se sont traduits par de très nombreuses variantes. Le modèle GRAFCET a été défini afin d'harmoniser les langages de description des automatismes logiques. Cependant ces représentations conduisent rapidement à des graphes de tailles conséquentes.

Ainsi, les STATECHARTS, développés sous l'impulsion de Harel [Harel 87], proposent un formalisme visuel plus expressif pour décrire des systèmes réactifs complexes. Ils résultent d'une extension des machines à états dont la structuration en macro-états parallèlisés et/ou hiérarchisés permet de maîtriser l'explosion combinatoire.

Cependant ces modèles font abstraction des aspects fonctionnels, et en particulier ils ne considèrent ni le flux de données (types des données manipulées, provenance, destination), ni le rôle et l'influence des fonctions sur ces données. Ces aspects sont appréhendés par les méthodes fonctionnelles.

▷ **Méthodes fonctionnelles:** La méthode SADT (Structural Analysis Design Technique), fournit une approche rigoureuse pour décrire et structurer une application. Par contre rien n'est prévu pour exprimer la logique de contrôle du système (synchronisations, parallélisations). La méthode SART (Structured Analysis for Real Time) comble en partie cette lacune en définissant un diagramme de flots de données qui englobe données fonctionnelles et signaux de contrôle. ARGOS [Maraninchi 90] repose exclusivement sur la manipulation d'événements sans possibilité de traitement de données mais en isolant cependant les composantes fonctionnelles.

Les ACTIVITY-CHARTS ([Harel 90]) présentent l'intérêt d'apporter un complément fonctionnel à la description comportementale des statecharts. Cette "combinaison" fait apparaitre des notions qui nous rapprochent du problème tel que nous nous le posons. Les activity-charts

décrivent les fonctions et les signaux ou les données qui circulent à l'intérieur du système ou entre le système et l'environnement. Cette description se fait en termes d'*actions* élémentaires considérées instantanées (lecture d'une donnée, émission d'un signal), hiérarchiquement composées en *activités* qui enchaînent des actions et/ou d'autres activités. Contrairement aux actions, une activité peut prendre du temps dans l'attente d'un signal. A chaque activité est associée un *statechart de contrôle* qui en exprime la décomposition en actions et sous-activités, ainsi que les transitions. Enfin, la structuration de l'ensemble du système est décrite par les *modules-charts* qui hébergent les activités et des bases de données. Les modules sont connectés via des *canaux*. La séparation entre les aspects fonctionnels et comportementaux d'une part, et la hiérarchisation des activités et des états d'autre part, apportent une grande lisibilité au système.

Cependant, si ces modèles et méthodes définissent des notions essentielles qu'il faut considérer lors de la conception de systèmes temps réel, ils s'éloignent de notre problématique. En effet, leur finalité est de guider la spécification et/ou de valider une application *définie a priori*, alors que notre problème est relatif à l'organisation et à l'implémentation d'un système distribué *multi-fonctionnels* dont la logique de contrôle est établie *en ligne* par un niveau décisionnel selon la tâche à accomplir *et* les données produites. Dans ce contexte, une modélisation exhaustive de toutes les combinaisons d'exécution parait irréaliste, d'autant que le système, dans sa relation avec l'environnement, est soumis à un flux *asynchrone* d'événements dont les instants d'occurence sont peu prédictibles. On retiendra cependant un certain nombre de principes relatifs à la logique de contrôle de sous-systèmes et à la spécification fonctionnelle.

2.1.3 Les systèmes réactifs organisés

Comme nous l'avons vu, le développement de systèmes réactifs pose tout un ensemble de problèmes nouveaux dans la conception de systèmes informatiques relatifs au temps réel et au parallélisme. Si des langages spécifiques sont proposés et si des opérateurs de base pour le parallélisme, le synchronisme ou le transfert de données entre des processus sont offerts par les systèmes d'exploitation, il n'existe pas de méthodologie générale, ni d'outils évolués pour structurer et intégrer ces systèmes. En partant de cette constatation plusieurs travaux ont été menés afin de proposer des structures générales et des contextes d'intégration. Nous présentons trois systèmes qui ont pu se traduire par des intégrations sur différents robots et dans différents cadres d'applications.

2.1.3.1 ControlShell

CONTROLSHELL, conçu à "Aerospace Robotics Laboratory" à Stanford University et développé conjointement avec "Real-Time Innovation, Inc." [Schneider 95], est un contexte d'intégration temps réel qui fournit des outils pour construire des composants fonctionnels partageables et spécifier leurs interactions par flux de données, et un exécutif programmable sous forme d'une machine à états finis qui supervise l'exécution du système constitué des différents composants.

Les auteurs ont volontairement privilégié la facilité d'intégration et de validation par l'expérimentation de systèmes faisant intervenir de nombreux traitements périodiques interconnectés, aux dépens d'une vérification formelle du comportement global du système (temps de traitement, inter-blocages, ...).

Les éléments de base sont des *modules* de programmes partageables et réutilisables. De nouveaux modules peuvent être ajoutés à une bibliothèque qui en comporte déjà un certain nombre prédéfinis (filtres, calcul de trajectoires, exécution de mouvements). Les *modules échantillonnés* sont les fonctions qui interviennent dans des traitements périodiques. A partir de ces modules sont construits, au moyen d'un éditeur graphique, des *composants* constitués d'un ou plusieurs modules échantillonnés incluant des fonctions d'initialisation et de terminaison, pour lesquels sont spécifiés toutes les données d'entrée et de sortie. Un éditeur de flux de données permet alors de connecter des composants, plus précisément des instances de ces composants, en des groupements qui définissent des sous-systèmes à flux de données. Les groupements actifs sont sélectionnés dynamiquement durant l'exécution, ce qui permet de rediriger le flot de données. Les stratégies d'activation/désactivation sont programmées sous forme d'un graphe d'états, les *modules de transition*, qui spécifie les actions à accomplir sous l'impulsion d'événements. Les relations entre les composants, les actions et les événements ne sont cependant pas très claires.

CONTROLSHELL a été utilisé dans de nombreuses applications relatives aux robots manipulateurs (coordination, contrôle par retour d'effort, contrôle de structures flexibles, contrôle adaptatif, ...).

CONTROLSHELL est un système très souple, aisément reconfigurable, et qui peut prendre en compte des événements asynchrones durant l'exécution de traitements périodiques complexes, mais dont la tâche globale reste, à un certain niveau d'abstraction, relativement simple. L'exemple présenté dans [Schneider 95] illustre bien le domaine d'application de ce système: il consiste en la préhension, par deux bras manipulateurs coordonnés, de pièces qui défilent sur un tapis roulant. L'opération s'effectue en quatre étapes en commutant successivement de stratégie de génération de trajectoire: approche du convoyeur en contournant les obstacles fixes modélisés par un système de vision, placement au dessus de la pièce selon les informations du convoyeur, saisie de la pièce par retour d'effort, puis placement coordonné de la pièce par les deux manipulateurs sous contrôle visuel. La tâche consiste donc en une succession d'actions prédéfinies.

2.1.3.2 Chimera

CHIMERA, issu d'une collaboration entre le Robotics Institute de CMU et l'Université de Maryland[Stewart 95b, Stewart 95a] est également un contexte d'intégration modulaire de systèmes temps réel. Les composants sont définis comme étant des "port-based objects": un objet standard qui inclut un état interne, du code et des données encapsulées et une interface constituée des ports d'entrée/sortie et de ressources. Les ports d'entrée/sortie sont utilisés pour connecter des objets d'un même sous-système alors que les ports de ressources sont dédiés à l'interaction avec un opérateur, une ressource physique ou un autre sous-système.

Chaque objet donne lieu à une tâche (informatique), ces tâches s'exécutent *indépendamment* les unes des autres (sans synchronisation) sur différents processeurs de l'architecture distribuée. Pour cela, chaque objet dispose d'une zone mémoire locale (les données transférées par les ports) qu'il rafraîchit et exporte, au fur et à mesure des besoins et de la production, dans une mémoire commune sans aucune synchronisation entre les tâches. Les objets ont quatre états: NOT-CREATED, ON, OFF et ERROR, qui transitent sur événement. Dans l'état ON, un cycle est exécuté à la réception d'un signal d'éveil périodique (horloge) ou

apériodique.

Un groupement de tâches est obtenu en connectant les ports des objets (configuration statique).

Les "jobs" (tâches robotiques) sont des séquencements prédéfinies de groupements de tâches autorisés par les capacités de reconfiguration en ligne du système. Bien que cela soit à l'étude, il ne s'agit pas réellement de reconfiguration dynamique car elle ne peut s'opérer que lorsque le robot est à l'arrêt. Un job est une application du type "prendre un objet", "explorer un objet par capteur tactile", "effectuer le suivi visuel d'un objet". Les jobs peuvent eux-mêmes être séquencés.

Une application particulièrement bien adaptée à l'utilisation de ce système à composants réutilisables est la commande du manipulateur modulaire reconfigurable RMMS de CMU[3]. CHIMERA est dédié à l'intégration et la coordination d'asservissements, pour différents types de manipulateurs, qui composent des tâches prédéfinies qui peuvent être sélectionnées en ligne. Ce système, focalisé sur l'efficacité des mécanismes de partage et de transfert de données multi-tâches et multi-processeurs, est construit sur un système d'exploitation également dénommé CHIMERA.

2.1.3.3 ORCCAD

Le système ORCCAD de l'INRIA [Simon 93] est également dédié à la conception et à l'intégration de systèmes robotiques mais avec une approche beaucoup plus formelle, fondée sur le langage synchrone ESTEREL, qui permet de procéder à des vérifications de propriétés temporelles et comportementales.

Un concept clef est l'action robotique qui est définie comme un asservissement associé à un comportement. Ainsi l'entité principale est la *tâche-robot* qui est:

- la spécification paramétrée d'un asservissement: une loi de commande de structure invariante sur la durée d'exécution de la tâche robot, intégrée dans des *fonction-tâches*.
- un comportement logique associé à un ensemble de signaux relatifs à l'action élémentaire d'asservissement et qui peuvent se produire
  - juste avant l'exécution: les *préconditions*,
  - pendant l'exécution: les *exceptions* qui sont de trois types:
    - type 1: la réaction se traduit par la modification de paramètres des lois d'asservissement,
    - type 2: la tâche-robot est tuée, elle devra être relayée par une autre tâche-robot,
    - type 3: erreur fatale, l'application est stoppée.
  - à l'issu de l'exécution: les *postconditions*.

La tâche-robot est constituée d'un ensemble de tâches temps réel communicantes, les *tâches-modules*, qui implémentent chacune une partie de la loi d'asservissement ou du comportement, et dont la majorité sont périodiques. La définition complète des tâches-module, et donc celle de la tâche-robot, nécessite la spécification de contraintes temporelles (durée et période) et de primitives de synchronisation. Six tâches-module sont consacrées à la gestion du comportement, dont l'une coordonne l'ensemble de la tâche-robot sous l'impulsion des signaux internes

3. Le manipulateur est assemblé en emboîtant des liens et des articulations mécaniques de différents types commandées par des moteurs indépendants.

et des signaux provenant d'un niveau supérieur de contrôle. Cette tâche-module, décrite par un programme ESTEREL, se traduit par un automate à états finis qui permet de valider un certain nombre de propriétés de la tâche-robot (inter-blocage, vivacité, sûreté). Les tâches-robot composent la couche fonctionnelle du système robotique et vont permettre d'élaborer des actions robotiques plus complexes.

Une application robotique s'exprime par des combinaisons logiques et temporelles des tâches-robot dont les interfaces sont des signaux typés[4]. Une action ainsi programmée est nommée *procédure-robot* [Kapellos 94]. Une procédure-robot comporte:

- un programme principal qui correspond à l'exécution nominale de l'action et qui est un arrangement logique de tâches-robot et de procédures-robot,
- un ensemble de triplets (événements d'exception, traitement, assertion) qui indiquent le traitement à effectuer à la réception de l'exception et l'information à transmettre à un éventuel niveau supérieur,
- un comportement local qui définit la coordination logique des éléments précédents.

Le programme principal est obtenu par des opérateurs exprimant les séquences, les parallélismes, les conditions, les itérations, les rendez-vous et les préemptions. La procédure-robot est traduite en ESTEREL et donne lieu à un certain nombre de vérifications formelles [Espiau 95].

ORCCAD est particulièrement adapté aux applications qui font intervenir de nombreux asservissements dont il faut coordonner et adapter les exécutions selon le plan d'action défini et les événements d'exception rencontrés.

L'application la plus démonstrative concerne le contrôle de cellules flexibles d'assemblage [Coste-Maniere 92]. Plus récemment le système a été employé dans le cadre de deux applications en robotique mobile: la première consiste en l'asservissement visuel (signal vidéo) d'un véhicule autonome sur un véhicule piloté dont le mouvement n'est pas uniforme; en cas de perte du signal vidéo le véhicule doit se garer [Kapellos 95]. La seconde application consiste à rejoindre une zone donnée en suivant un mur par asservissement visuel et en surveillant la présence d'obstacles, y localiser une cible, puis se placer devant celle-ci [Pissard-Gibollet 95].

2.2 Les approches architecturales

L'étude des architectures de contrôle décisionnel pour robot mobile autonome, privilégie l'analyse des besoins et fonctionnalités pour attribuer au robot des capacités de raisonnement afin d'accomplir des tâches dans un environnement mal connu et évolutif.

Ces travaux, multiples, ont donné lieu à de nombreuses classifications et analyses comparées dans lesquelles sont généralement opposées les approches hiérarchiques et les approches distribuées et leurs qualités décisionnelles ou réactives. Ce schisme doit être modulé car il semble maintenant bien établi dans la communauté scientifique, que l'autonomie implique un ensemble de processus *réactifs* proches des capteurs et des actionneurs *et* un niveau supérieur *décisionnel centralisé* qui ne peut en effet raisonner de façon cohérente que s'il dispose, à un certain niveau d'abstraction, d'informations globales concernant la tâche et l'état du système. Les niveaux décisionnels peuvent être multiples et, selon la complexité de la tâche, peuvent faire appel à des techniques relevant de la planification d'actions.

4. Par exemple, pour exécuter séquentiellement deux tâches-robot, il suffit d'exprimer que les postconditions de synchronisation de la première satisfont les préconditions de synchronisation de la seconde.

La frontière entre le réactif et le décisionnel ne se délimite cependant pas par les différents sous-systèmes de l'architecture: les processus réactifs doivent agir de façon *cohérente* pour tendre vers l'objectif fixé et le niveau décisionnel doit être *réactif* face aux changements qualitatifs qui peuvent se produire à tout instant et impliquer des actions de reprise et/ou une remise en cause globale du plan.

L'élaboration d'une architecture de contrôle consiste donc à trouver un compromis entre distribution et centralisation des compétences afin de concilier deux notions a priori antinomiques: décision et réaction. Les architectures vont se distinguer par le nombre de sous-systèmes, leur niveau de compétence, la nature des données manipulées et les relations qu'ils entretiennent avec les autres sous-systèmes.

Nos propres choix architecturaux ont été exposés au cours du premier chapitre. Nous ne présentons ici que les approches les plus significatives dans la mesure où elles proposent des concepts généraux qui se sont traduits par de nombreuses intégrations et variantes. De très nombreuses études présentent les différentes architectures, en particulier Haseman [Hasemann 95] fait un point très détaillé des nombreuses approches et Degallaix [Degallaix 93] les expose selon une classification par compétence (exécutif, superviseur, affineur, planificateur).

Parmi les nombreuses approches, nous en citons quatre qui sont représentatives de démarches ou d'écoles de pensées.

▷ **les approches comportementales** En s'inspirant de l'analyse du comportement animal, l'éthologie, Brooks défend la théorie selon laquelle on peut construire des agents intelligents[5] sous la forme d'une hiérarchie de comportements purement réactifs, sans contrôle centralisé ni représentations sur lesquelles raisonner [Brooks 85, Brooks 91]. Selon ce précepte, il définit une architecture de "subsomption" où chaque niveau implémente un comportement sous la forme d'un réseau fixe de modules qui peuvent inhiber les entrées ou les sorties de modules du niveau immédiatement inférieur (figure 2.1 page suivante). Cette approche a donné lieu à de nombreuses expérimentations qui ont effectivement démontré des capacités comportementales intéressantes.

En fait la planification des tâches est codée "en dur" et se révèle particulièrement efficace lorsque l'objectif se résume à un comportement (explorer sans heurter d'obstacles, aplanir un terrain par mouvements aléatoires, aller vers la lumière, ramasser des boîtes de soda) qui ne nécessite ni représentation ni raisonnement, et lors qu'aucune situation non nominale ne risque de se présenter.

Bien que personne ne nie le besoin d'actions réflexes, en particulier pour assurer la sécurité du véhicule, cela ne suffit pas à l'accomplissement cohérent de tâches "utiles". D'ailleurs, la capacité de prédire l'effet de ses actions grâce à des modèles est aussi un critère de sécurité (*e.g.* éviter une zone dangereuse).

▷ **Les architectures organisées autour d'un contrôleur central.** Tout à fait à contre-pied des approches purement réactives, les approches centralisées telles que l'architecture TCA (Task Control Architecture) [Simmons 90] développée à CMU (figure 2.2 page suivante), organisent un nombre arbitraire de modules autour d'un module de contrôle central.

5. La notion d'intelligence est définie par les auteurs comme une mesure de l'adéquation avec un environnement et non plus comme une faculté d'analyse. Ce qui leur permet entre autres de qualifier d'intelligent un automate [Miller 93].

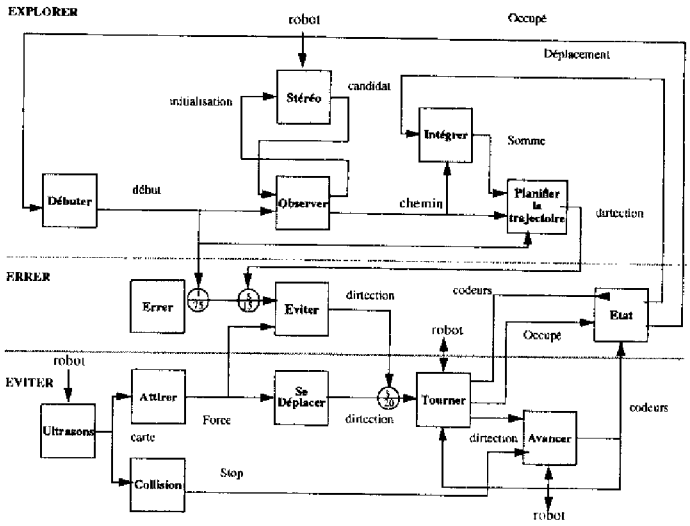


FIG. 2.1 – *L'architecture purement réactive par "subsomption".*

Les modules sont chargés d'exécuter des processus de traitement locaux tels que la perception, la planification de chemins et l'exécution. Le contrôleur central gère les ressources, décide des actions et transmet les messages entre les modules. Toutes les communications transitent par le contrôleur qui les redistribue aux modules concernés. Cette hyper-centralisation limite les interactions entre les modules et donc la possibilité de programmer des actions réflexes, et peut conduire à un goulet d'étranglement. TCA a été intégré sur différentes plateformes expérimentales: contrôle de la marche du robot d'exploration à six pattes AMBLER, contrôle des robots mobiles d'intérieur HERO (chargé de ramasser des tasses au moyen d'un bras manipulateur) et XAVIER qui explore l'environnement, et du prototype de robot d'exploration lunaire RATLER [Simmons 95].

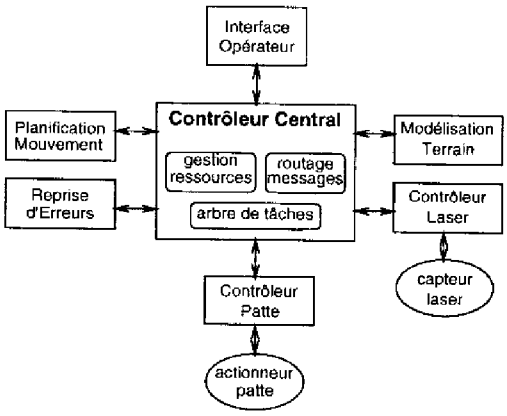


FIG. 2.2 – *L'architecture centralisée TCA.*

▷ **Les architectures strictement hiérarchiques.** Ces architectures cherchent à concilier décision et réactivité: en partant de l'hypothèse que la dynamique du monde décroît avec le niveau d'abstraction, l'architecture NASREM proposée par Albus [Albus 87] est une décomposition hiérarchique selon laquelle chaque tâche peut se décomposer en sous-tâches qui vont constituer le niveau immédiatement inférieur (figure 2.3 page suivante). Ainsi, aux niveaux les plus bas se trouvent les traitements rapides tels que les asservissements qui manipulent

des données brutes ou peu affinées. Et aux niveaux supérieurs des processus lents de raisonnement, tel que le planificateur de tâches, utilisent des données affinées essentiellement symboliques. L'architecture complète comporte six couches qui sont elles-mêmes décomposées horizontalement en trois parties: perception/modélisation/action ou planification. Cette structuration très stricte a le mérite d'être claire mais sa rigidité en fait un système parfois difficile à adapter au problème considéré. Cette architecture, sous sa forme plus récente RSC, a été intégrée dans des applications variées [Albus 95].

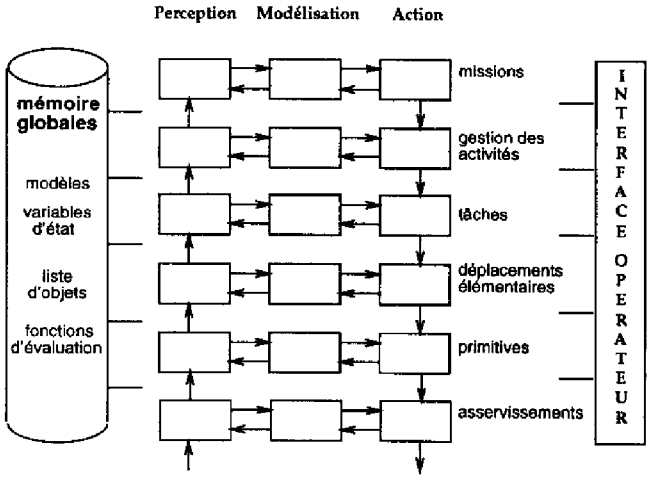


FIG. 2.3 – *L'architecture hiérarchique NASREM.*

▷ **Architecture développée au LAAS:** cette architecture a été explicitée au chapitre précédent. Rappelons en deux mots qu'elle est structurée en deux parties offrant différents niveaux de réactivité et d'abstraction: une couche fonctionnelle distribuée chapeautée par une couche décisionnelle.

2.3 Conclusion

▷ **Les langages** temps réel constituent un grand pas vers la formalisation de systèmes réactifs. Les différents langages que nous avons présentés s'inspirent plus ou moins directement de la logique temporelle et sont, d'un point de vue logique, équivalents. Ils se traduisent à la compilation par un automate à états finis à partir duquel des méthodes et des outils de vérification du comportement logique ou de contraintes temporelles ont pu être élaborés. L'hypothèse synchrone permet de raisonner sur des horloges logiques et les contraintes temporelles se traduisent par des relations de précédence ou de simultanéité et des durées exprimées en nombre d'opérations élémentaires; la durée "vraie" dépend quant à elle du processeur.

Cependant, ces langages diffèrent par leur mode d'expression et par là même s'adaptent plus ou moins naturellement à certaines classes de problèmes. Deux classes de problèmes se prêtent bien à cette approche et peuvent concerner les applications robotiques:

- *les systèmes à flot de données*, tels que les circuits et les filtres numériques, qui impliquent des calculs parallèles périodiques se décomposant naturellement en actions élémentaires pouvant être considérées instantanées. Le système peut alors être *complètement* exprimé par un langage synchrone et donner lieu à une analyse approfondie, voire exhaustive, de son comportement logique et temporel. L'horloge est une horloge

logique (ou un ensemble d'horloges logiques) qui marque chacune des transitions (ou chaque cycle - Rex). Le système étant entièrement défini, des méthodes d'intégration automatiques peuvent être élaborées à partir de la description de l'architecture matérielle (*e.g.* SYNDEX). Application en robotique: traitement d'images.

- *les systèmes de contrôle d'exécution* d'actions externes au système de contrôle pour lesquelles le langage synchrone permet d'exprimer le séquencement logique, sous l'impulsion d'événements produits par ces actions ou par des ordres issus d'un niveau de contrôle supérieur. Les actions sont généralement exprimées dans un autre langage plus général. Les événements provenant de processus qui évoluent indépendamment du système de contrôle, l'automate doit être activé par une machine d'exécution qui les réceptionne et les synchronise. Cet automate devra répondre, et donc réagir, en un temps compatible avec la production des événements et pour cela les techniques d'optimisation ou de scission d'automates sont un atout. L'intérêt de ce modèle est également de pouvoir vérifier la logique du système: non blocage, complétude, *etc.*, mais aussi aide à l'analyse fonctionnelle relative au cahier des charges (adéquation, sécurité, *etc.*). Par contre, à moins de connaître parfaitement les propriétés temporelles des actions (*e.g.* périodiques de durée constante), il est quasiment impossible de prouver la satisfaction de contraintes temporelles. Application en robotique: contrôle de systèmes ou de sous-systèmes.

Cependant, les approches synchrones, dans le cadre de l'intégration de fonctions de traitement, restent fondamentalement dédiées aux processus périodiques: les filtres et les asservissements. C'est pourquoi ces systèmes se révèlent particulièrement adéquats pour les applications dont le problème principal réside dans l'intégration de boucles d'asservissement rapides, sensibles (stabilité), reconfigurables dynamiquement et interconnectées, telles que la coordination des axes d'un ou de plusieurs bras manipulateurs.

En robotique mobile autonome, ce problème d'intégration de boucles d'asservissement est moins sensible car les asservissements périodiques sont en nombre réduit[6], avec des interactions lâches entre eux (non synchronisés) et des algorithmes strictement séquentiels et invariants, donc de durée constante. Le seul point critique concernant cet aspect est alors la durée du traitement relativement à sa période; cependant, cette durée étant invariante, elle est aisément mesurable et les seules réponses envisageables en cas de dépassement sont l'optimisation de l'algorithme ou l'utilisation d'un processeur plus puissant.

Par contre, le robot mobile autonome nécessite aussi une variété de processus dont les temps de traitement sont très variables et souvent apériodiques. C'est pourquoi l'utilisation de langages génériques, associés à une structuration formelle, semble mieux adaptée dans ce cas précis.

▷ **Les systèmes réactifs structurés.** Un autre aspect qui ressort de cette analyse est l'indispensable *structuration* du système (*i.e.* organisation des actions, du ou des contrôleurs, des interactions) et la nécessité de méthodologies d'intégration, ce que les langages ne permettent pas d'exprimer.

C'est pourquoi doivent être définies des méthodes de spécification structurées hiérarchiques et des contextes d'intégration, même si ceux-ci ne doivent pas reposer exclusivement

6. Au plus bas niveau intervient un asservissement en position dont la consigne peut être produite sur la base d'un asservissement sur données proximétriques et/ou d'un asservissement visuel.

sur une approche synchrone. Il n'existe pas de méthodes générales, mais de l'analyse de différentes solutions émergent différents concepts et exigences que nous retrouverons dans le système G^{en}oM que nous avons développé: distribution modulaire de fonctions de traitement, reconfigurabilité statique et dynamique, interactions événementielles et à flots de données efficaces, contrôle global du système.

Ainsi le système présente une structuration hiérarchique comportant un *niveau fonctionnel*, qui intègre les différents traitements (actions) à mettre en œuvre, chapeauté par un *contrôleur d'exécution* (figure 2.4):

- Le niveau fonctionnel est structuré en *entités de traitement* (*e.g.* STATECHARTS: activités, CONTROLSHELL: composants, CHIMERA: port-base objects, ORCCAD: tâches-robot, G^{en}oM: modules[7]) où doivent clairement se distinguer:
  - *les fonctions* de traitement (*e.g.* STATECHARTS: actions, CONTROLSHELL: modules, CHIMERA: code des port-base objects, ORCCAD: fonctions-tâche, G^{en}oM: codels).
  - *la structure* ou contexte d'exécution des fonctions qui interagissent par transfert de données (*e.g.* STATECHARTS: activity-charts, CONTROLSHELL: modules-habitats, CHIMERA: tâches, ORCCAD: tâches-module, G^{en}oM: tâches d'exécution).
  - *le comportement* ou la logique de contrôle cadencée par des événements synchrones ou asynchrones (*e.g.* STATECHARTS: statechart de contrôle, CHIMERA: graphe d'état, ORCCAD: tâche-module de coordination, G^{en}oM: tâche de contrôle)[8].
- Le contrôleur d'exécution coordonne les traitements au moyen d'événements selon un plan préétabli et selon l'évolution ou les états des actions marqués également par des événements. (*e.g.* STATECHARTS: statechart de contrôle, CONTROLSHELL: modules de transition, CHIMERA: groupements de tâches séquencés par des jobs, ORCCAD: procédures-robot, G^{en}oM: rôle joué dans notre démarche de manière plus générale par un exécutif codé par exemple avec Kheops ou PRS.).

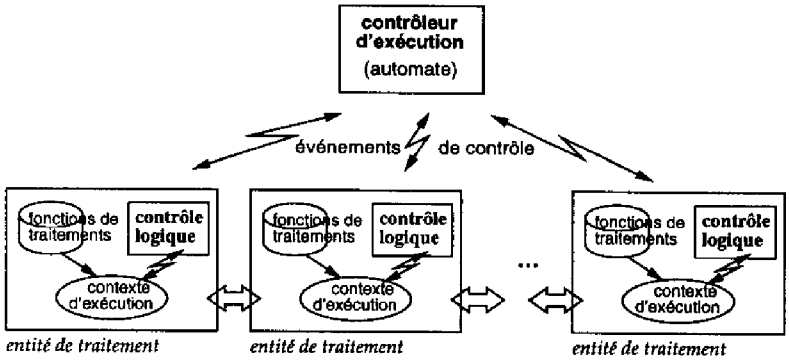


FIG. 2.4 – *Organisation "classique" des systèmes distribués temps réel.*

Une structuration formelle est de façon générale indispensable pour définir un système réactif. Cependant, le contrôle d'un robot mobile autonome ne peut se réduire au séquencement de sous-systèmes réactifs au moyen d'un automate à états finis. L'évolution de son environnement et la variété des situations qu'il peut rencontrer en font également un système décisionnel qui doit pouvoir élaborer des comportements complexes, choisir la manière

7. La description précise des concepts de G^{en}oM sera fournie dans le chapitre 2 de la partie II.

8. CONTROLSHELL agit uniquement par changement dynamique de composants et par redirection du flot de données requis à un niveau supérieur.

dont il exécute ses tâches, mettre en œuvre des traitements calculatoires complexes (apériodiques), construire des modèles de représentation à différents niveaux d'abstraction sur lesquels des traitements purement calculatoires (*e.g.* planification de trajectoires), mais aussi des raisonnements sur des données symboliques (*e.g.* stratégie de déplacements), pourront être élaborés.

Cela implique, au niveau de la couche fonctionnelle:

- de pouvoir produire et exporter des données plus ou moins affinées dans des bases de données structurées auxquelles pourra accéder toute autre entité[9];
- de pouvoir construire des comportements plus élaborés grâce au contrôle direct d'entités de la couche fonctionnelle sur d'autres entités, conduisant à des *hiérarchies dynamiques* de comportement (par opposition à des comportements "à plat", sous le contrôle direct de l'exécutif - figure 2.4 page précédente-). Cette notion sera illustrée par la suite par un arbre dynamique d'activités dont la racine est le contrôleur d'exécution ou *exécutif*. Par exemple, un processus de suivi visuel ou de suivi de trajectoire peut requérir en environnement contraint un processus d'évitement local qui lui-même contrôlera l'asservissement en position.
- Enfin, si un contrôle logique réactif de la couche fonctionnelle est nécessaire, par exemple au moyen d'une machine à états finis, ses interactions avec le niveau décisionnel ne peuvent se résumer aux seules relations événementielles. En effet, le système robotique *n'évolue pas uniquement sous l'influence de ses propres actions*, mais aussi en fonction des variations intrinsèques de son environnement qui est ouvert[10]. Les changements qualitatifs induits par ces variations devront être signalés et des données structurées devront pouvoir être transférées du niveau de traitement vers le niveau de décision (*e.g.* état du robot, nouveaux obstacles, cartographies établies à partir de données perceptuelles, chemins planifiés, *etc.*), mais aussi du niveau supérieur vers le niveau inférieur (*e.g.* modification du modèle topologique de l'environnement, modèles d'amers, trajectoires d'autres robots dans le cadre de planification multi-robots, *etc.*).

Cette analyse met en valeur les différences et les complémentarités des approches de type génie logiciel des systèmes distribués temps réel, et les approches architecturales qui soulignent les contraintes et spécificités des systèmes décisionnels réactifs. Nous pensons que la conception d'architectures de contrôle pour robot mobile autonome ne peut aboutir sans considérer à tous les niveaux les problèmes mis en exergue par l'une et l'autre approche.

Ainsi, dans la seconde partie de ce mémoire consacrée à la couche fonctionnelle, nous insisterons sur le rôle de cette couche dans l'architecture, et en particulier sur ses interactions avec le niveau décisionnel et sa composante temps réel distribuée. Ces différentes considérations ont permis une spécification précise de la couche fonctionnelle et de ses composants qui s'est traduite par une méthodologie générale de développement et d'intégration des traitements qui la composent.

9. CHIMERA permet ce type de transfert mais le limite à des données de taille réduite.

10. Dans une cellule flexible d'assemblage, le modèle de l'environnement évolue sous l'unique influence des actions du manipulateur (*e.g.* prise/dépose de pièce) [Kapellos 94]. En robotique mobile autonome l'environnement évolue indépendamment du robot et peut conduire à des changements de l'état de celui-ci (*e.g.* obstacle).

Deuxième partie

La couche fonctionnelle modulaire

Chapitre 1

La couche fonctionnelle

La couche fonctionnelle constitue l'interface entre le niveau décisionnel et le monde physique dans lequel évolue le robot, entre la décision et l'action. C'est grâce à elle que les données numériques issues de la perception vont devenir des données symboliques interprétables par des processus de raisonnement de plus haut niveau, que les tâches requises vont se traduire en actions, et que le robot maintiendra une interaction permanente avec son environnement: elle est le lieu des asservissements et des actions réflexes. Ces aspects fonctionnels sous-tendent déjà un certain nombre de propriétés: l'observabilité, la programmabilité et la réactivité.

La difficulté d'intégration des fonctions qui vont composer la couche fonctionnelle provient de leur disparité, tant par le type des données manipulées que par leurs contraintes temporelles, et de la complexité des algorithmes mis en œuvre. Ces fonctions vont traiter du contrôle des capteurs et des actionneurs, de la modélisation et de l'interprétation de données sensorielles, de la planification de trajectoires, *etc.*, requérant des savoir-faire spécifiques.

Malgré tout, ces fonctions devront présenter une certaine uniformité comportementale et organisationnelle afin d'être intégrées dans l'architecture de contrôle du robot, en s'abstrayant de l'algorithmique sous-jacente. En effet, elles ne prendront pleinement leur sens que lorsqu'elles pourront être contrôlées directement ou indirectement, par un opérateur ou par un système décisionnel afin d'accomplir la tâche attribuée au robot.

Nous allons dans un premier temps exhiber les *propriétés* nécessaires de cette couche fonctionnelle qui en ont guidé la conception. Une représentation de son *comportement*, ou de son activité globale, sera exposée en termes d'*activités*. Nous présenterons alors une *structuration* en *modules* qui permet d'organiser la couche fonctionnelle en y regroupant les activités. Les interactions au sein de cette couche fonctionnelle s'opèrent selon deux modes qui permettent respectivement d'en contrôler les activités et de rendre accessible des données fonctionnelles ou d'état: *le flux de contrôle* et *le flux de données*. La structuration et le fonctionnement interne des modules seront l'objet du chapitre suivant.

1.1 Propriétés et caractérisation

Le robot, son état, ses actions, ses interactions avec l'environnement et l'observation de cet environnement sont appréhendés et contrôlés, par le niveau décisionnel ou par un opérateur, au travers de la couche fonctionnelle. Cette couche fonctionnelle met en œuvre des

boucles d'asservissement, des actions réflexes, des opérations de modélisation et de planification. Celles-ci sont invoqués à un plus haut niveau par un opérateur ou par un superviseur d'exécution et tendent globalement à l'accomplissement d'une tâche ou d'une mission tout en garantissant l'intégrité du système. Pour permettre le contrôle de ces exécutions, la couche fonctionnelle doit fournir une observabilité permanente de l'état et des résultats des traitements.

Ainsi, la couche fonctionnelle se présente simultanément comme:

- une bibliothèque de services,
- un contexte d'exécution temps réel,
- une image de l'activité et de l'état du robot,
- une représentation de l'environnement perçu par le robot,

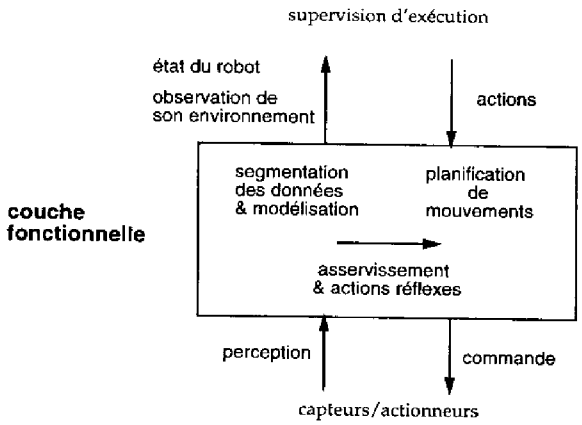


FIG. 1.1 – *Le rôle de la couche fonctionnelle*

Ces multiples fonctions rendent nécessaire une analyse des propriétés et caractéristiques qui ont orienté nos choix conceptuels. Les trois grands aspects qui vont caractériser cette couche fonctionnelle sont la dynamique du système qu'elle contrôle, son rôle de serveur vis à vis d'un niveau décisionnel et le contexte d'intégration de fonctions qu'elle représente.

1.1.1 Un système dynamique

Un robot mobile est avant tout un système physique qui maintient une relation permanente avec un environnement réel. La dynamique propre du système (mouvements), mais également la dynamique relative aux interactions avec l'environnement, induite par le mouvement ou due à l'évolution de cet environnement (obstacles mobiles, mouvements asservis sur des données capteurs, ...), impliquent des capacités temps réel de la couche fonctionnelle. En particulier, elle doit offrir des fonctions de base spécifiques aux systèmes d'exploitation temps réel telles que l'interception d'événements asynchrones, la parallélisation et la commutation de tâches en temps borné.

▷ **Maîtrise des temps de traitement** La maîtrise de la dynamique du système implique la maîtrise du temps d'exécution des processus d'asservissement. Il faudra donc disposer de moyens pour contrôler la disponibilité d'une ressource particulière qu'est le temps CPU.

▷ **Réactivité** Le robot doit être capable de prendre en compte des événements asynchrones et d'y réagir en des temps qui soient compatibles avec la dynamique du système.

▷ **Adaptabilité** Il devra pouvoir opérer des changements de mode d'exécution en fonction de l'évolution du contexte (changement de mode d'asservissement, d'acquisition, de modélisation, ...).

1.1.2 Propriétés relatives à l'aspect décisionnel

Les diverses tâches que le robot sera amené à exécuter sont des combinaisons d'actions qui seront sélectionnées et dont les paramètres seront instanciés selon l'état du robot et l'observation de son environnement. Les conséquences au niveau de la couche fonctionnelles (CF) sont les suivantes:

▷ **Programmabilité** Les différents services offerts par la CF doivent pouvoir être sélectionnés selon la tâche à accomplir et les conditions d'exécution. La CF doit donc être programmable et permettre des exécutions séquentielles, parallèles ou conditionnelles d'actions, leur interruption ou encore des modifications de configuration de ces actions.

▷ **Observabilité** Afin d'être contrôlable à un plus haut niveau, la CF doit rendre disponible en permanence une représentation de l'activité globale du robot, c'est à dire des actions de la CF en cours d'exécution. L'accès à des informations caractérisant l'état du robot (*e.g.* niveau des batteries, alarmes) ainsi que ses connaissances sur son environnement, déduites de données sensorielles, est également nécessaire.

▷ **Robustesse** Il faut disposer, au niveau de la CF, de moyens pour détecter et analyser les cas de défaillances et surtout pour installer des fonctions de reprises ou des modes dégradés afin de ne pas remettre en cause l'ensemble de la mission.

1.1.3 Un contexte d'intégration

Le robot expérimental est en permanence en développement. D'autre part, la réalisation de robots réels doit s'appuyer sur la réutilisabilité de composants logiciels. Au delà d'un contexte d'exécution, la CF doit être un contexte d'intégration:

▷ **Intégrabilité** La CF doit être organisée de façon à rendre aisée l'intégration cohérente d'une nouvelle fonction en guidant l'installation de celle-ci, en donnant des moyens de validation, et en la rendant disponible sous forme d'un service à un plus haut niveau (opérateur, niveau décisionnel).

▷ **Evolutivité** Afin d'augmenter les capacités opérationnelles du robot, la CF doit pouvoir s'enrichir incrémentalement de nouvelles fonctionnalités sans remettre en cause celles qui sont déjà présentes. Cela doit permettre non seulement d'apporter des services originaux au niveau de la CF, mais également de définir des traitements élaborés qui font eux-mêmes appel à des services préexistants, simplifiant d'autant les programmes à un plus haut niveau.

▷ **Reconfigurabilité** La validation ou la mise au point sera plus aisée si la CF offre des moyens pour modifier des paramètres d'une fonction, voire sélectionner telle ou telle fonction, durant le déroulement d'une application.

1.2 L'activité de la couche fonctionnelle: Les activités

La couche fonctionnelle est un ensemble de fonctions qui peuvent être activées, désactivées ou reconfigurées. On appelle *activité* une fonction en cours d'exécution. Une fonction peut avoir, à un instant donné, plusieurs instances actives, correspondant à autant d'activités. L'ensemble des activités présentes à un instant donné représente l'activité globale du système robotique ([Chatila 90]).

On peut distinguer trois moyens d'action, qui peuvent être combinés, d'une activité sur le système robotique:

- **par production de données.** *e.g.* calcul de trajectoires, segmentations d'images.
- **par actions sur le robot logique.** *e.g.* commande des capteurs et des actionneurs.
- **par activation d'autres activités.** *e.g.* mouvements avec asservissement visuel, avec évitement d'obstacles.

Cette dernière possibilité est essentielle car elle va permettre d'obtenir des comportements plus élaborés par des combinaisons de comportements élémentaires. On peut ainsi définir des activités qui procéderont elles-mêmes à des activations ou désactivations conditionnelles d'activités, à des exécutions parallèles ou séquentielles. C'est un des facteurs d'*évolutivité* et également de *réactivité* du système robotique. De façon générale, toute tâche pré-programmable, c'est à dire dont les activités sont connues (mais pas leurs séquencements ni leurs paramètres d'exécution) peut être exprimée sous la forme d'une activité composée.

1.2.1 Arbre d'activités

Lorsqu'une activité fait appel à d'autres fonctions offertes par le système, celles-ci produisent à leur tour des activités. Ces nouvelles activités sont dites *activités filles* de l'activité qui les a créées, elle même nommée *activité mère*. Ainsi, l'ensemble des activités présentes à un instant donné constitue un *arbre d'activités* (figure 1.2) représentant l'activité (au sens générique du terme) globale du robot.

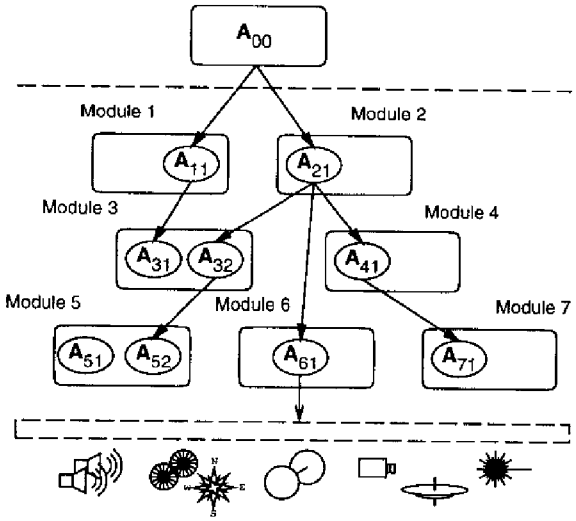


FIG. 1.2 – *Arbre d'activités.*

Notons qu'à la racine de l'arbre nous n'avons pas à proprement parler une activité telle que celles qui sont présentées dans ce paragraphe. Il peut s'agir soit de l'exécutif d'un niveau

décisionnel (voir le chapitre précédent), soit d'une interface homme/machine.

Il est important de remarquer que cet arbre est dynamique: à chaque instant des activités peuvent se terminer et d'autres peuvent démarrer. L'arrêt d'une activité n'est effectif que lorsqu'elle n'a plus d'activité fille. Ainsi, l'interruption d'une activité entraîne nécessairement l'interruption de ses activités filles. Toute activité est interruptible en un temps compatible avec la dynamique du système contrôlé.

Nous avons donc une hiérarchie d'activités où l'on distingue les *activités composées* qui font appel à des sous-activités et les *activités terminales* aux extrémités de l'arbre. Ces dernières sont également dénommées *activités primitives* lorsqu'elles ne font jamais appel à des sous-activités mais qu'elles agissent exclusivement soit sur le robot logique, soit en produisant des données.

Certaines activités doivent être présentes en permanence dès l'initialisation du système. Ces *activités permanentes*, peu nombreuses, concernent pour l'essentiel l'asservissement en position du véhicule mais qui peuvent prendre d'autres formes selon l'application considérée. Ces activités ininterruptibles sont directement sous le contrôle de l'activité à la racine de l'arbre.

1.2.2 Exemple d'activité composée

Afin d'illustrer cette arborescence considérons l'exemple suivant: on souhaite faire exécuter au robot un mouvement gardé, autrement dit exécuter une trajectoire sans heurter d'obstacles. La première activité concernera le calcul de la trajectoire désirée. Cette activité terminée, deux autres se dérouleront en parallèle: l'exécution de la trajectoire et la surveillance de présence d'obstacles (figure 1.3). Ces activités se redécomposent elles-mêmes en sous-activités. Ainsi, l'exécution de la trajectoire fera appel à deux activités filles: le calcul de consignes et l'asservissement sur ces consignes.

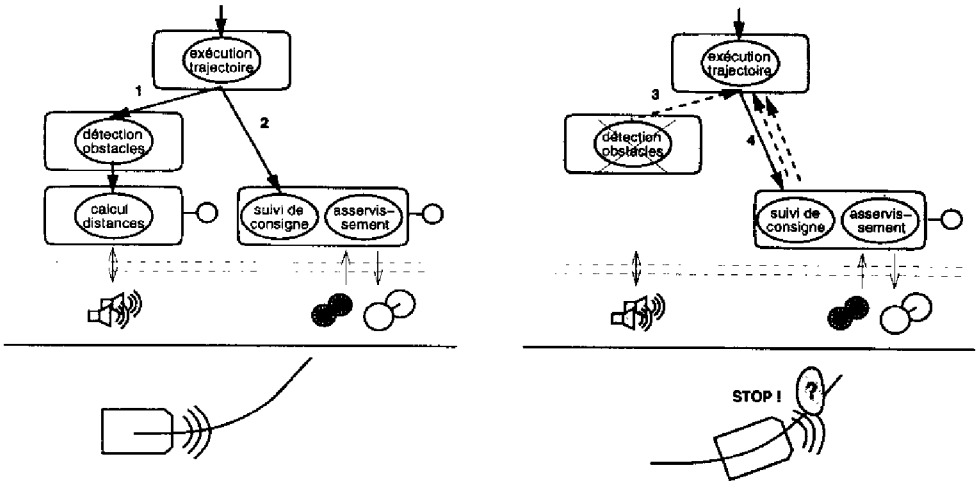


FIG. 1.3 – *Exemple d'action réflexe.*

Le déclenchement de la surveillance d'obstacles devra immédiatement être suivi d'une action d'évitement de l'obstacle (par contournement ou arrêt du robot). Contrairement au

séquencement d'activités *"calcul de trajectoire/exécution de trajectoire"*, l'enchaînement *"détection d'obstacle/évitement d'obstacle"* doit se faire dans un laps de temps compatible avec la dynamique du système (vitesse du robot selon la distance aux obstacles). L'évitement d'obstacle consistera donc en un enchaînement conditionnel d'activités. L'activité composée ainsi réalisée exprime une *action réflexe* (< condition >⇒< action >) pour laquelle le temps de transition entre la détection de la condition et l'action sera négligeable et ne dépendra pas du temps de réaction de l'exécutif (en particulier si le contrôleur d'exécution est directement un opérateur!). On constate à cette occasion que les activités composées permettent aussi de développer des services d'aide à la commande "manuelle" (par un opérateur) du robot.

A chaque instant l'arrêt du véhicule et donc celui de l'activité d'exécution de trajectoire peut se révéler nécessaire. De façon générale, la réactivité et l'adaptabilité du système requièrent l'*interruptibilité* des activités. Cependant, dans le cadre du contrôle d'un système réel, cette interruption ne peut se traduire par la suppression brutale du traitement comme on le ferait en simulation. Une activité interrompue devra transiter préalablement par une phase de stabilisation du système contrôlé. En reconsidérant l'exemple ci-dessus, l'interruption de l'exécution d'une trajectoire impliquera l'exécution d'une trajectoire d'arrêt, l'interruption d'une activité de suivi de consigne impliquera le calcul d'une consigne d'arrêt, l'interruption de l'activité d'asservissement (déconseillée) coupera les moteurs. Plus l'interruption sera requise à un haut niveau dans l'arborescence des activités, mieux celle-ci sera maîtrisée mais plus le temps de réaction sera long (à rapprocher de la hiérarchie de réactions dans la conduite automobile: lever le pied, freiner, piler). L'interruptibilité des activités sera plus amplement étudiée dans le chapitre suivant.

1.2.3 Classification des activités

Les fonctionnalités embarquées à bord d'un robot mobile autonome sont très diverses et en particulier du point de vue du contrôle, les caractéristiques temporelles, les conditions d'arrêt, les interactions avec d'autres activités et les flux de données diffèrent. Les activités associées à ces fonctions ont été classées en quatre catégories:

- **les asservissements** sont des activités périodiques structurellement invariantes[1] qui maintiennent une interaction "forte" du système avec l'environnement. En mode nominal elles ne se terminent pas d'elles-mêmes. Les données de sortie, fonctions de données d'entrée issues de capteurs logiques éventuellement filtrées, sont les consignes d'un effecteur logique. Exemples: asservissement en position, asservissement visuel, asservissement par ultrasons, *etc.*
- **les filtres** sont également des activités périodiques structurellement invariantes. Ils exportent des données filtrées à la fréquence de l'activité. Selon le type de filtre les données d'entrée seront accédées périodiquement ou transmises intégralement à l'initialisation du filtre. Dans le premier cas le filtre devra être arrêté explicitement et dans le second il se terminera de lui-même avec la fin de la consommation des données d'entrée. Exemples: calcul de position par odométrie, génération de points de consignes le long d'une trajectoire (auto-terminante), *etc.*

1. *i.e.* dont les algorithmes déroulent toujours les mêmes séquences d'instructions (complexité d'ordre 0), et par conséquent dont le temps d'exécution est invariant.

- **les surveillances** sont des activités structurellement invariantes qui peuvent être périodiques (surveillances par scrutation) ou apériodiques (attente d'un signal), et qui se terminent à l'occurrence de l'événement attendu. Exemples: détection d'obstacles par capteurs proximétriques, de changement de lieu topologique, de changement de mode d'exécution, d'alarme *etc.*
- **les serveurs** sont des activités asynchrones qui se terminent à l'issu du traitement. Selon la complexité des algorithmes déroulés, la maîtrise du temps de traitement peut être délicate. Celle-ci est généralement moins cruciale que pour les trois autres types d'activité; elle devra cependant rester compatible avec la tâche à effectuer. Exemples: Calcul de chemin géométrique, extraction de primitives perceptuelles, contrôle de l'exécution d'une trajectoire, *etc.*

Les activités vont donc être essentiellement caractérisées par les attributs suivants:

- périodique ou apériodique
- auto-terminante ou à terminaison contrôlée
- primitive ou composée
- accès à des données durant l'exécution

La majorité des activités primitives vont concerner la commande des capteurs et des actionneurs et seront donc de type asservissement ou filtre. Les comportements plus complexes seront généralement obtenus par des combinaisons d'activités gérées par une activité de type serveur. Ainsi le contrôle de l'exécution d'une trajectoire fait appel à des filtres et des asservissements et se termine avec la fin de la trajectoire (ou l'occurrence d'une erreur). On peut également citer des activités primitives de type serveur dont le rôle est purement calculatoire et dont les contraintes temporelles sont plus lâches (*e.g.* recherche de chemin topologique, modélisation de terrain).

1.3 Un réseau de modules

La diversité des fonctions embarquées et la flexibilité requise au niveau de la couche fonctionnelle plaident pour une organisation systématique des fonctions, des activités et des interactions. C'est pourquoi les fonctions ont été regroupées en modules chargés du déroulement des activités. Ces regroupements s'opèrent selon des critères de partage de ressources (physiques ou données), de contraintes en temps de traitement (caractéristiques temps-réel compatibles), ou lors d'interactions fortes (en temps ou données) entre des activités. En pratique cela reste assez largement à la discrétion du concepteur du module, la finalité étant de proposer un sous-système complet et indépendant offrant une homogénéité de services et des interfaces bien définies.

Ainsi, un module encapsule une bibliothèque de fonctions de traitement, les activités qui mettent en œuvre ces fonctions, des données exploitées et/ou produites par ces activités et des mécanismes de contrôle d'exécution (voir la figure 1.4 page 41).

Cette structuration en modules est un point fondamental de la couche fonctionnelle (CF). Elle lui confère de nombreuses propriétés énoncées au début de ce chapitre. En dix points, elle permet:

- de décharger l'exécutif en délocalisant une partie de la gestion des activités au niveau des modules.

- de rendre la CF plus robuste, indépendamment de l'exécutif, en traitant localement les procédures d'interruption ou de reprise locale.
- d'attribuer au système un caractère évolutif par adjonction, suppression ou remplacement de modules.
- d'offrir aux spécialistes un cadre d'intégration pour incorporer ou modifier des algorithmes sans remettre en cause les interfaces du module avec le reste du système.
- de partager, au niveau du module, des bibliothèques de fonctions de traitement et des bases de données.
- de proposer pour chaque module une interface de contrôle standard, en termes de services, pour démarrer, arrêter ou configurer les activités.
- de définir des moyens normalisés de transfert de données structurées.
- d'exploiter au sein d'un module des procédures offertes par les OS temps réel particulièrement efficaces en mono-processeur (un module est une entité qui ne peut être répartie sur plusieurs processeurs).
- de guider la répartition de la CF, en termes de modules, entre différents processeurs.
- de proposer des outils communs de validation et de mise au point.
- de découpler les problèmes de mise au point et d'augmenter la "lisibilité" de la CF.

Tous ces points seront plus amplement exposés et démontrés au cours du mémoire.

Contrairement à d'autres types d'organisation, il n'y a pas au niveau de la couche fonctionnelle de couches hiérarchiques prédéfinies. Chaque module est directement accessible. Ceci autorise des interventions directes par un opérateur à n'importe quel niveau, et surtout, cela permet au superviseur de réaliser des tâches qui ne sont pas nécessairement déjà définies sous la forme d'un service d'un module.

Ainsi, un module est un *serveur*[2] qui propose un certain nombre de services ainsi que des accès à des données exportées. Comme nous allons le voir par la suite, ces deux types d'interaction (services et données) s'opèrent par le biais de deux protocoles de communication distincts ([Ferraz De Camargo 91]) qui gèrent respectivement *le flux de contrôle* et *le flux de données* entre les modules comme cela est illustré par la figure 1.4 page suivante.

La figure 1.5 page ci-contre présente un exemple de couche fonctionnelle du robot *Hilare2* et les relations potentielles entre les modules. Cette couche fonctionnelle sera détaillée par la suite.

1.4 Le flux de contrôle: les requêtes

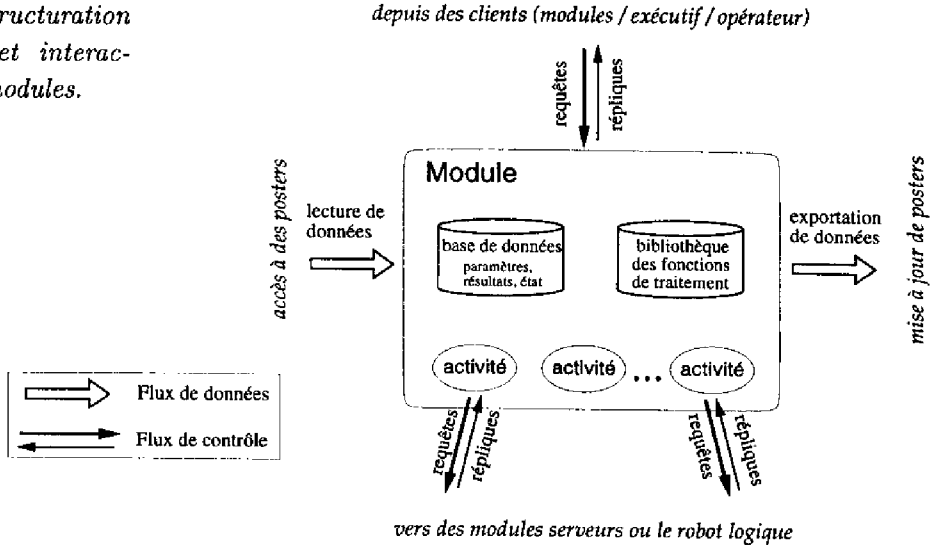
On accède aux différents services offerts par la couche fonctionnelle, c'est-à-dire par les modules, au moyen de *requêtes*. Les requêtes permettent de démarrer, d'interrompre ou d'altérer des traitements. Les requêtes vont en particulier cadencer l'évolution de l'arbre d'activités de la couche fonctionnelle.

Les requêtes peuvent être émises par n'importe quel élément du système (un autre module, l'exécutif, une interface-opérateur). La réception de la requête par le module établit une relation de type *client/serveur* (figure 1.6 page suivante).

Un service se termine par l'émission en retour, du serveur vers le client, d'une *réplique finale*, ce qui a pour conséquence de clore la relation client/serveur.

2. A ne pas confondre avec les activités de type serveur.

FIG. 1.4 – *Structuration en module et interactions inter-modules.*



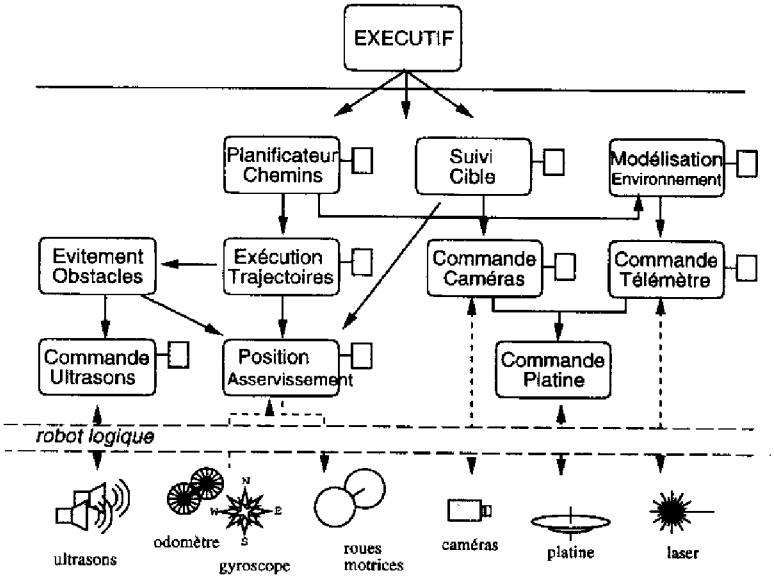
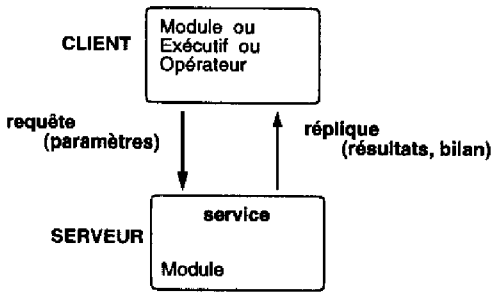


FIG. 1.5 – *Couche fonctionnelle du robot Hilare2.*

FIG. 1.6 – *Une relation client/serveur.*



Le caractère temporel de la relation client/serveur répond à un concept de base de la structure de la couche fonctionnelle: *Pour que le système soit évolutif, un module serveur ne doit pas avoir connaissance a priori de ses clients.* Des modules peuvent être ainsi incrémen-

talement ajoutés à la couche fonctionnelle, et utiliser des services déjà définis sans remettre en cause les modules préexistants.

Lorsqu'une activité d'un module requiert, par une requête, une activité auprès d'un autre module, alors cette nouvelle activité est fille de la précédente et leurs modules établissent une relation client/serveur. Toutes les activités d'un module sont, selon une image généalogique, d'une même "génération" (*i.e.* sœurs ou cousines). Au même titre qu'une activité ne peut être simultanément mère et fille d'une autre activité, un module ne peut être simultanément client et serveur d'un autre module. Ce principe évite des bouclages qui rendraient le contrôle des activités inextricable.

On peut associer à une requête des *paramètres d'exécution*. Les répliques finales peuvent retourner des *résultats d'exécution*. Chaque réplique est porteuse d'un *bilan d'exécution* qui qualifie la façon dont le traitement s'est déroulé. Il signale en particulier les exécutions non-nominales, les interruptions d'activité, la porté du résultat retourné. Ces bilans d'exécution seront plus largement présentés dans la section 1.4.2. Les paramètres transmis par une requête sont enregistrés par le module dans une base de données interne les rendant ainsi accessibles à l'ensemble des éléments du module et en particulier aux activités (figure 1.7). La robustesse du module repose en partie sur la cohérence de cette base de données. Le module doit donc disposer de moyens pour contrôler la validité de ces paramètres et éventuellement en refuser l'enregistrement, ce qu'il signifiera au moyen du bilan d'exécution.

Enfin, l'émission de requête et la réception de répliques sont des opérations non bloquantes afin de permettre la parallélisation des traitements.

1.4.1 Deux types de requêtes

Chaque requête est typée et associée à un service donné d'un module. Ce service peut être le démarrage d'une activité, ou des changements de configuration du module (paramètres d'exécution). Ces deux aspects donnent lieu à deux types de requêtes distincts: *les requêtes d'exécution* et *les requêtes de contrôle* (figure 1.7).

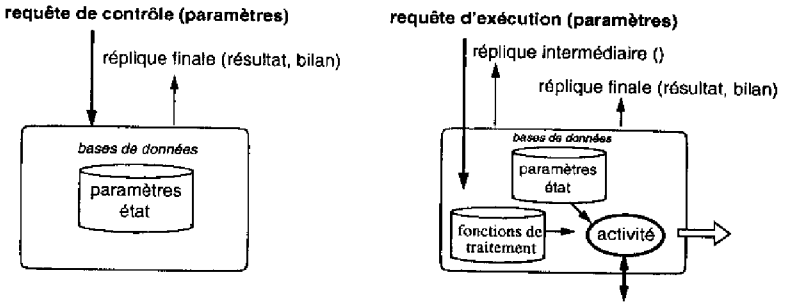


FIG. 1.7 – *Les deux types de requête.*

1.4.1.1 Les requêtes d'exécution

La réception d'un requête d'exécution est associée au lancement d'une activité. La réplique finale sera retournée au client à la fin de cette activité.

Le démarrage d'une activité n'étant pas, en cas de conflits avec des activités déjà présentes, toujours immédiat (temps d'interruption des activités incompatibles), une *réplique intermédiaire* est émise vers le client pour lui signifier le début effectif du traitement. Contrairement

aux répliques finales, les répliques intermédiaires ne retournent pas de bilan d'exécution, mais un *identificateur d'activité* qui permettra de la référencer pour accéder à des informations la concernant ou pour l'interrompre explicitement.

Certaines fonctions de traitement peuvent avoir plusieurs instances actives à un instant donné, donnant lieu à autant d'activités (*e.g.* des surveillances avec différentes valeurs attribuées aux conditions de déclenchement, concaténations de trajectoire). Ces fonctions doivent bien évidemment être déclarées compatibles avec elles-mêmes. Chaque requête de ce type créera alors une nouvelle activité sans interruption des précédentes.

Le bilan de la réplique finale caractérise la façon dont s'est déroulé le traitement et en particulier il signale une éventuelle interruption.

1.4.1.2 Les requêtes de contrôle

Contrairement aux requêtes d'exécution, les requêtes de contrôle ne démarrent pas d'activité. Elles ont pour fonction d'accéder à des données d'exécution, de modifier des paramètres d'exécution ou d'interrompre une activité. Elles ne procèdent elles-mêmes à aucun traitement: le service associé est considéré de durée nulle. La réplique finale est par conséquent retournée immédiatement au client.

▷ **Accès à des données:** Un moyen simple d'accéder aux données d'un module est de les demander par une requête de contrôle et de les réceptionner dans la réplique. Si cela est bien adapté à des accès asynchrones et occasionnels, on verra cependant qu'il existe un moyen beaucoup plus efficace qui sera présenté dans la section 1.5 consacrée au flux de données.

▷ **Modification de paramètres:** Les requêtes peuvent transmettre des paramètres qui sont alors enregistrés dans la base de données du module. Selon la façon dont le traitement a été conçu, ces nouvelles valeurs peuvent être prises en compte *dynamiquement* par une activité en cours, ou lors de l'initialisation d'une prochaine activité. On peut ainsi modifier "en ligne" des paramètres d'exécution tels que les gains d'un asservissement, les distances d'évitement ou les vitesses de parcours d'une trajectoire, ou encore apporter des changements plus fondamentaux au mode d'exécution d'une activité tels que des commutations de loi d'asservissement ou de source de données proprioceptives (odométrie/gyroscope) utilisées dans le calcul de la position du véhicule.

▷ **Interruption d'activité** Tout comme une requête d'exécution, une requête de contrôle peut interrompre une activité. Cette interruption peut être obtenue soit par l'effet de bord des incompatibilités, soit en invoquant une requête de contrôle prédéfinie nommée `abort` qui permet d'interrompre explicitement une activité en la référençant par son numéro d'identification. L'arrêt effectif de l'activité sera signifié dans le bilan de la réplique finale de cette dernière. Notons que l'interruption d'une requête de contrôle n'a pas de sens puisque son traitement est de durée nulle.

1.4.2 Erreurs et bilans

Un robot mobile est un système dynamique qui évolue dans un monde physique: les exécutions en mode nominal sont fréquemment remises en cause et le caractère autonome du système rend les cas de dysfonctionnement particulièrement cruciaux. Il faut donc disposer de moyens pour détecter au plus tôt et pour interpréter les signes de défaillance des activités.

Chaque service se solde par une réplique qui inclut un compte-rendu d'exécution: le bilan d'exécution. En mode nominal, ce bilan une valeur par défaut qui indique que l'exécution s'est déroulée normalement. Dans le cas contraire, il permet de caractériser le dysfonctionnement.

Ce qui importe c'est de déterminer si la mission peut se prolonger (éventuellement en mode dégradé) ou, durant les phases de mise au point, d'identifier la cause du dysfonctionnement afin d'y remédier. On distingue trois types de bilans selon leurs conséquences sur le système:

- Une information qui est sans conséquence pour le serveur qui concerne le client ou le superviseur:
  - détection de paramètres de requêtes invalides.
  - interruption d'un service.
  - échec d'un service. (*e.g.* Impossibilité de trouver un chemin ou de localiser un amer, perte de cible).
- Une défaillance du serveur qui nécessite une intervention externe. (*e.g.*Véhicule en mode manuel, choc de l'arceau de sécurité).
- Une erreur fatale qui rend le service ou l'ensemble du module temporairement ou définitivement inutilisable (*e.g.* Erreur système, capteur en panne, situation sans procédure de reprise prévue).

Le premier de ces trois types de défaillance peut être repris par le système autonome. Dans certains cas, une redondance matérielle ou logicielle pourrait parer à une panne définitive d'un capteur ou d'un serveur (troisième type de défaillance) à condition que la panne soit bien identifiée. A ce titre, il est très important d'estimer l'ensemble des défaillances qui peuvent se produire et de déterminer les procédures de reprise associées: une défaillance imprévue ne peut être corrigée de façon automatique.

Dans tous les cas, le système doit dans la mesure du possible se stabiliser, et retourner au plus tôt une information complète et pertinente à l'opérateur ou au système de contrôle distant. Chaque bilan intègre, outre l'information spécifique, les références du module et de l'activité concernée.

1.4.3 Réactivité et conflits

Le système garantit que *toute requête émise est réceptionnée et traitée par son destinataire.* Qui plus est, pour assurer *la réactivité* du système, chaque requête est traitée au plus tôt et dans l'ordre d'arrivée, qui conserve lui-même l'ordre d'émission par un client donné. En effet, le module ne dispose pas à son niveau de moyens pour analyser la pertinence de telle ou telle requête, il doit par contre réagir au plus vite à toute demande.

Il n'y a par conséquent *pas de priorité entre les requêtes.* En cas de conflits la plus récente prévaut toujours. Ce choix, guidé par le critère de réactivité, permet d'uniformiser le comportement des modules par une règle simple aisément assimilable par un exécutif. Dans le cadre d'une application intégrée, les conflits doivent être globalement résolus par le niveau décisionnel. Cela implique que les conflits soient prédictibles; il est par conséquent impératif de se doter de moyens pour de *déclarer les incompatibilités entre les services* d'un module.

Cependant, un serveur pouvant avoir plusieurs clients simultanément, la *robustesse* du système exige que les conflits soient également correctement gérés au niveau du module. La requête la plus récente prévalant toujours, les activités incompatibles antérieures sont interrompues. Cela présuppose que toute activité soit interruptible en un laps de temps

compatible avec la dynamique du système contrôlé par le module (ce qui n'implique pas que l'interruption soit instantanée).

Le contrôle local de l'interruption des activités est, dans la mesure où il est correctement géré, d'une grande efficacité dans un contexte temps réel, car il permet d'interrompre directement une activité sans transiter par les activités mères successives. On peut ainsi mettre en place des procédures d'arrêt d'urgence. Les activités mères en seront informées par la réception des répliques finales.

1.5 Le flux de données: les posters

Si le protocole client/serveur est adapté au problème du contrôle des activités et de la configuration d'un module, il ne convient pas à la publication ou au transfert de données entre activités. Ce protocole impose en effet:

- une relation hiérarchique (temporaire) entre les modules qui ne se retrouve pas nécessairement dans le flux de données. En effet, des activités sans lien "parental" direct peuvent avoir à échanger des données.
- une synchronisation de type "rendez-vous" entre des activités dont les contraintes temporelles diffèrent.

Un protocole de transfert de données adapté au *flux de données* a donc défini, indépendamment du flux de contrôle. Il permet à chaque module d'exporter des données dans des bases de données structurées externes nommées *posters*. Ces posters sont accessibles en lecture à tout élément de l'architecture de contrôle du robot: les autres modules mais également l'exécutif ou un opérateur (figure 1.8).

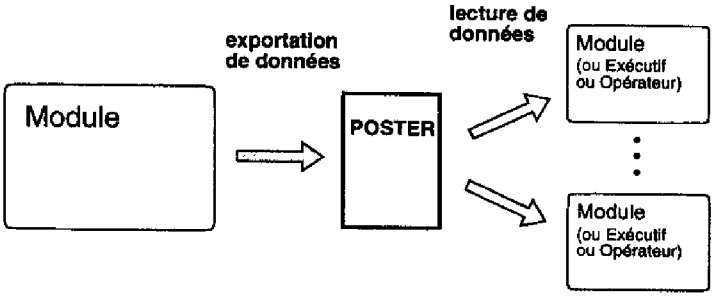


FIG. 1.8 – *Le flux de données: les posters.*

Ce protocole est simple et efficace car seul le module propriétaire du poster peut en modifier le contenu. On accède directement aux données du poster (à la structure complète ou à des sous-structures) au rythme souhaité indépendamment du module propriétaire.

Ces posters vont permettre de rendre publiques des données statiques volumineuses (les données de taille réduite peuvent être accédées par requêtes de contrôle), de disposer des données maintenues à jour au fur et à mesure de leur génération, de transférer des données entre des activités indépendamment de leur caractéristiques temporelles ou encore de procéder à des transferts synchrones si les activités émettrice et réceptrice sont périodiques.

Les posters ne contiennent pas uniquement des données fonctionnelles issues des activités, mais également des informations concernant l'état du module et de ses activités. On distingue ainsi deux types de posters: *les posters de contrôle* et *les posters d'exécution* (figure 1.9 page suivante).

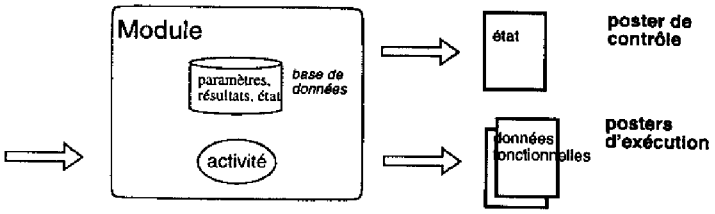


FIG. 1.9 – *Poster de contrôle et posters d'exécution.*

▷ **Les posters de contrôle** ont été définis afin de connaître à chaque instant l'état courant des modules. A chacun d'entre eux est alloué un poster de contrôle. Ces posters ont une structuration standard qui inclut la liste des activités, leur type, leur constante temporelle, leurs durées d'exécution théorique et réelle, leur stade d'exécution, les identités des clients et des serveurs, ... L'*observabilité* de la couche fonctionnelle est en grande partie réalisée par les posters. La connaissance de l'état de chacune des activités va également concourir à la mise au point du système.

▷ **Les posters d'exécution** sont des posters spécifiques aux traitements mis en œuvre et sont mis à jour par les activités du module. Ci-dessous sont précisées les principales utilisations de ces posters:

- **Centralisation de données contextuelles constantes.** Cet aspect est important car cela permet de paramétriser les algorithmes, et ce de façon cohérente pour tout le système (*e.g.* dimensions du véhicule, position des capteurs, modèles d'amers, cartes topologiques, *etc.*). L'initialisation de ces posters peut être effectuée par l'exécutif ou par une activité spécifique, par exemple à partir d'un fichier de données dont les références sont transmises en paramètre de la requête.
- **Exportation de données "semi-statiques"** ou plus précisément produites à l'issu d'une activité de type serveur. Ce sont généralement des données "lourdes" (*i.e.* denses ou issues d'un traitement long) qui sont ainsi maintenues disponibles à l'ensemble du système, et ce dès la fin de l'exécution (image de points 3d, chemin géométrique, modèle numérique de terrain, *etc.*).
- **Transfert de données instationnaires.** Ce sont des données qui évoluent en temps réel sous l'impulsion d'activités de type filtre, surveillance ou asservissement. Ces données sont ainsi rafraîchies à la fréquence de cette activité (position courante du véhicule, consigne d'asservissement, données proximétriques, *etc.*).

Chaque poster est référencé de façon unique. L'accès à tel ou tel poster par une activité est ainsi paramétrable (en transmettant la référence du poster en paramètre de la requête).

Un exemple illustratif de transfert de données entre activités est l'asservissement du véhicule sur une consigne instationnaire. La consigne n'aura pas la même origine selon que le véhicule est asservi sur une trajectoire prédéfinie, le long d'un obstacle découvert au fur et à mesure du mouvement par des données proximétriques ou encore en suivant une cible localisée par une caméra. Pour sélectionner le type d'asservissement adapté à l'application, il suffira alors d'indiquer en paramètre de la requête d'asservissement l'identificateur du poster du générateur de consignes souhaité.

La figure 1.10 montre un exemple de redirection du flux de données: on peut imposer à l'activité d'asservissement `''asservissement''` de s'asservir sur une consigne instationnaire `ref` produite soit par un générateur de trajectoire, soit par une activité de suivi visuel de

cible, ou soit encore par une activité de suivi de mur à partir de données proximétriques echos. Ces dernières pourront elles-mêmes être obtenues à partir de capteurs ultrasons ou d'un télémètre laser.

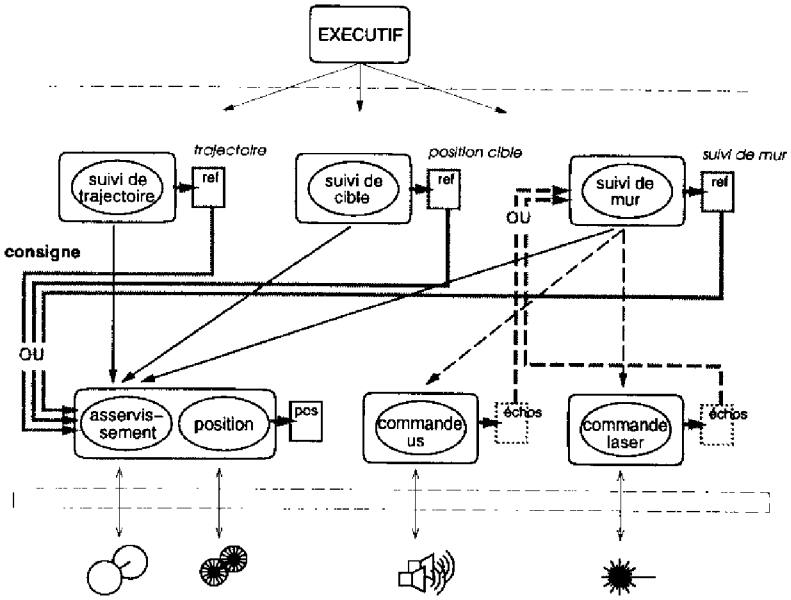


FIG. 1.10 – **Exemple de redirection de flux de données.** *Traits gris pleins: différentes origines possibles pour la consigne* ref, *traits gris pointillés: différentes origines possibles de données proximétriques* échos.

On constate que cette capacité de rediriger le flux de données augmente encore la flexibilité et la programmabilité du système.

Chapitre 2

Anatomie d'un module

La diversité des fonctions qui constituent la couche fonctionnelle, son évolutivité (intégration de nouvelles fonctions), et sa programmabilité (combinaison de ces fonctions) appellent à une structuration en *modules* afin de mieux appréhender chacune de ces fonctions. La description des modules est l'objet de ce chapitre.

Chaque module gère une ressource particulière en déroulant des algorithmes spécifiques. Le terme "ressource" est pris ici au sens large: il peut s'agir de capteurs ou d'actionneurs logiques ou de données structurées telles qu'une trajectoire, une image *etc.*. Les algorithmes pourront donc concerner des boucles d'asservissement, des procédures de surveillance ou des calculs complexes de segmentation, d'identification ou de modélisation, avec dans certains cas, de fortes contraintes de temps.

Cette structuration en modules doit, d'une part, permettre à un spécialiste -qu'il soit automaticien, mathématicien, ou versé dans le traitement d'images- d'ajouter aisément de nouvelles fonctions sans se préoccuper des aspects structurels; et d'autre part offrir un canevas qui respecte des règles de comportements et de structuration afin de s'intégrer dans l'architecture, indépendamment des aspects algorithmiques.

Un module devra donc combiner une description *comportementale* (états et transitions des activités), *fonctionnelle* (algorithmique des traitements et paramètres) et *structurelle* (intégration des traitements et interactions) du sous-système dont il a la charge.

Après une définition générale d'un module, nous allons analyser le cycle de vie d'une activité et les moyens de contrôle du module sur celle-ci. Nous ferons alors le lien entre les activités et les algorithmes qu'elles mettent en œuvre. Des données et des événements vont transiter au sein du module, ces interactions marquant la continuité, à l'échelle du module, des flux de contrôle et de données. Enfin, nous décrirons la structure logicielle complète d'un module.

Nous verrons tout au long de cette présentation comment se conjuguent, au niveau des modules, les propriétés essentielles de la couche fonctionnelle précédemment énoncées: programmabilité, réactivité, robustesse, observabilité, évolutivité et accessibilité; nous soulignerons éventuellement leur pertinence en nous appuyant sur des exemples concrets.

2.1 Définition d'un module

Les modules sont les briques de base de la couche fonctionnelle. Ils sont le lieu de l'exécution de l'ensemble des activités qui s'y déroulent. A ce titre, ils ont la charge:

- d'interagir avec les clients,
- de contrôler le déroulement des activités,
- d'exécuter les fonctions de traitement,
- de gérer les données internes,
- d'exprimer l'état global du module.

On distingue le système de contrôle et les activités contrôlées. Ces deux aspects seront présentés tour à tour dans les sections 2.2 "Contrôle d'une activité" et 2.3 "Activité et fonctionnalité" de ce chapitre. Le module est donc structuré en deux parties: le **niveau de contrôle** qui gère les activités en fonction de requêtes, et le **niveau d'exécution** où s'exécutent ces activités. Les différents éléments interagissent par le biais de bases de données dont les changements d'état sont signifiés par des événements de synchronisation. Les données relatives au contrôle des activités et à l'état du module et les données relatives aux traitements composent deux bases de données distinctes: *la Structure de Données Internes de Contrôle* ou SDI/c, et *la Structure de Données Internes Fonctionnelle* ou SDI/f (parfois nommée SDI d'exécution). La figure 2.1 schématise les fonctions d'un module et son organisation.

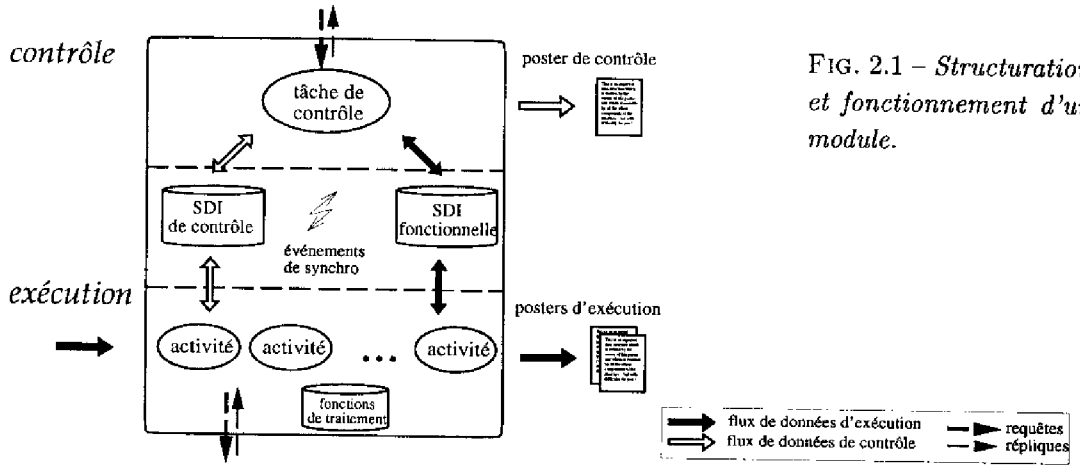


FIG. 2.1 – *Structuration et fonctionnement d'un module.*

2.2 Contrôle d'une activité

Les activités sont supervisées par le *contrôleur de module*. Ce contrôleur a, vis-à-vis de ces activités, quatre fonctions principales:

- **Réceptionner** les requêtes d'exécution, vérifier la validité de leurs paramètres.
- **Démarrer** les activités quand toutes les préconditions sont satisfaites.
- **Interrompre** les activités en cas de requête explicite d'interruption ou de conflit.
- **Transmettre** les résultats du traitement et le bilan d'exécution à l'activité mère.

▷ **Activation d'une activité** De la rapidité de déclenchement d'une activité dépend la *réactivité* du système. La prise en compte de la requête d'exécution est instantanée mais le

démarrage effectif du traitement peut, en cas de conflit, être différé. Dans ce cas, en appliquant la règle de priorité de la requête la plus récente, il devra dans un premier temps interrompre les activités antérieures incompatibles.

▷ **Interruption d'une activité** Le contrôleur peut interrompre toute activité, qu'elle soit auto-terminante ou pas. C'est un impératif pour pouvoir prendre en compte des changements d'état qualitatifs asynchrones. Cette interruption est demandée soit explicitement par une requête de contrôle spécifique, soit implicitement en cas de conflit.

Une interruption n'est pas, en général, instantanée. Selon le type de ressource gérée ou des spécificités algorithmiques, une interruption soudaine pourrait dans certains cas rendre la fonctionnalité inutilisable par la suite, ou pire encore, amener le système robotique dans un état instable et incontrôlé.

L'activité va donc transiter par un état intermédiaire (qui peut être de durée nulle) qui lui permettra de conclure son traitement. Elle devra en particulier interrompre d'éventuelles activités filles, stabiliser et caractériser l'état du système et réinitialiser des données en vue d'une prochaine activation.

▷ **Terminaison d'une activité** Le contrôleur doit détecter au plus tôt la fin d'une activité et la signaler à l'activité mère. On peut distinguer quatre cas de terminaison:

1. La terminaison normale d'activités auto-terminantes.
2. La terminaison suite à un échec prévisible sans conséquence sur l'intégrité du module.
3. La terminaison suite à une interruption.
4. La terminaison suite à une défaillance grave.

Du point de vue du contrôle on ne distingue pas les deux premiers cas. En effet, dans la mesure où la cohérence globale est maintenue et que cela n'influe pas sur le comportement du module, le contrôleur n'est pas concerné. Par contre, d'un point de vue fonctionnel, cette information devra être transmise au client par l'intermédiaire du bilan d'exécution associé à l'activité.

La terminaison à la suite d'une interruption est également signalée par un bilan d'exécution, sans autre conséquence sur le module.

La terminaison à la suite d'une défaillance grave doit être signalée au plus tôt au client. Comme pour les cas d'interruption, le système doit être si possible stabilisé et surtout il faut éviter toute propagation du problème. Pour cela, et afin de cerner le dysfonctionnement, l'activité peut être temporairement figée: exceptionnellement, une réplique finale qui, par son bilan va signaler cet état, est transmise avant la conclusion de l'activité. L'opérateur ou le superviseur pourra, selon son analyse, libérer cette activité par la requête particulière `abort`. Aucune nouvelle activité ne peut être démarrée dans un module dont une activité a été figée tant que l'`abort` n'a pas été demandé explicitement.

2.2.1 Graphe de contrôle d'une activité

Ces différentes étapes dans la vie d'une activité peuvent être représentées par un graphe d'état: *le graphe de contrôle d'une activité* (figure 2.2 page suivante). Les nœuds du graphe correspondent à des états de l'activité qui sont généralement de durée non nulle, et les arcs, à des transitions sur des événements considérées instantanées (c'est à dire en un temps compatible avec la constante temporelle de l'activité).

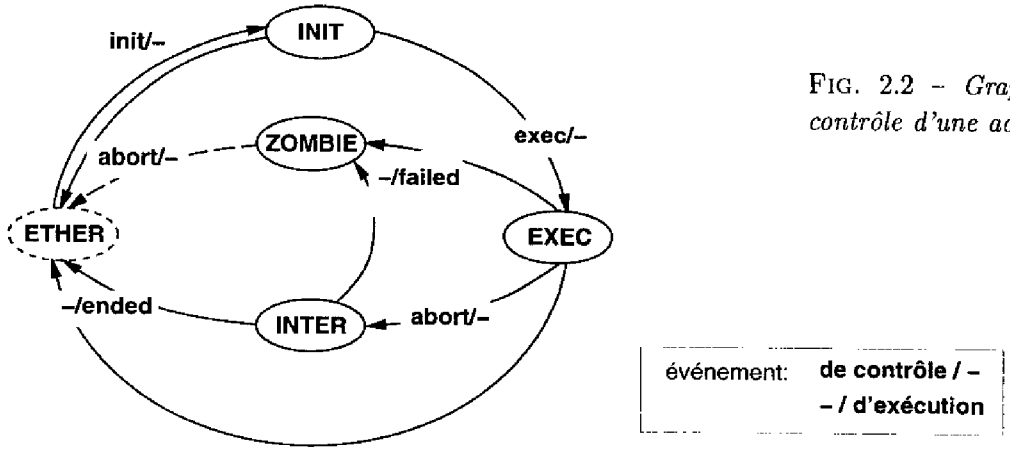


FIG. 2.2 - *Graphe de contrôle d'une activité.*

Le graphe de contrôle comporte cinq nœuds, correspondant à des états qui ont une durée (éventuellement nulle), reliés par neuf arcs correspondant aux transitions instantanées imposées par le contrôleur (marquées **cntrl/-**) ou signifiées par l'activité (marquées **-/exec**). On est initialement dans l'état fictif **ETHER**.

- A la réception d'une requête d'exécution, le contrôleur amène l'activité dans l'état **INIT** (**init/-**) durant lequel les conditions d'applicabilité de la requête sont vérifiées. C'est en particulier durant cet état que les éventuelles activités antérieures incompatibles sont interrompues. A ce stade, une interruption de l'activité (**abort/-**) la ramènerait directement à l'état **ETHER**.
- Lorsque toutes les préconditions sont satisfaites, le contrôleur requiert le démarrage effectif de l'activité auprès du niveau d'exécution au moyen de l'événement **exec/-** ce qui amène dans l'état **EXEC**.
- Si l'activité se termine d'elle-même (activité auto-terminante ou terminaison prématurée), elle le signifie par l'événement **-/ended** et revient à l'état **ETHER**.
- Si l'activité se termine d'elle-même suite à une erreur grave qui nécessite l'intervention d'un opérateur, elle le signifie par l'événement **-/failed** qui l'amène dans l'état **ZOMBIE**. La présence d'une activité dans cet état a pour effet de figer le module en interdisant toute nouvelle requête d'exécution. On ne peut revenir à l'état **ETHER**, et donc débloquer le module, que sur une requête de contrôle spécifique (**abort/-**). Cet état n'intervient normalement que lors de la mise au point du système.
- Si, suite à une requête de contrôle spécifique ou à une requête d'exécution incompatible, l'activité est interrompue (événement **abort/-**), elle transite dans l'état **INTER** durant lequel elle doit terminer au plus tôt son traitement[1]. Selon la façon dont se déroulera cette opération, elle reviendra alors à l'état **ETHER** (**-/ended**), ou elle se figera dans l'état **ZOMBIE** (**-/failed**).

1. Cette phase de terminaison permet de conclure proprement le traitement: stabiliser le système commandé, libérer des ressources allouées, terminer des activités filles, ...

2.3 Activité et fonctionnalité

Nous venons de décrire les étapes de la vie d'une activité *du point de vue du contrôleur* et les moyens d'action de ce contrôleur sur l'activité. Nous allons maintenant en considérer l'aspect fonctionnel, c'est à dire la façon dont elle exécute les fonctions de traitement. Ces traitements peuvent être de types très divers, tant d'un point de vue algorithmique que par des aspects temporels ou de séquencement. Ainsi certains d'entre eux auront un caractère répétitif ou devront invoquer des sous-activités. Les instants d'interruption possibles vont dépendre du traitement. Ce que nous proposons ici c'est d'offrir un contexte d'intégration le plus général possible afin d'aider le concepteur à intégrer sa fonction, et de rendre cette fonction immédiatement disponible dans la couche fonctionnelle.

2.3.1 Décomposition fonctionnelle d'une activité

Une activité est le déroulement d'un traitement donné. Ce traitement est défini sous la forme d'une fonction exécutable (une portion de code) qui peut éventuellement être invoquée plusieurs fois, en particulier dans le cadre d'activités périodiques. Le corps du traitement est généralement précédé d'une phase de démarrage (initialisation de sous-activités, de données, ...); et, pour les activités auto-terminantes, suivi d'une phase de terminaison (arrêt de sous-activités, libération de zones mémoires, ...). L'interruption fera elle-même appel à un traitement spécifique. En cas de défaillance ou tout au moins de changement de mode d'exécution, des variantes de la fonction de traitement peuvent s'avérer utiles.

De façon générale, une activité se décompose en différentes phases d'exécution qui font chacune référence à une fonction. Ces fonctions sont des portions de code *ininterruptibles* nommées *codel* (code élémentaire).

Cette ininterruptibilité ne relève pas d'un choix conceptuel mais d'une réalité physique: lorsque l'on contrôle un système ayant des interactions avec un environnement qui progresse indépendamment du système temps réel et de façon non-coopérante on ne peut se permettre d'interrompre à n'importe quel stade le processus de commande sans risque de dégradation. C'est au système de contrôle de se plier à la dynamique du processus contrôlé (fondamentalement asynchrone) et non pas l'inverse. De façon plus pragmatique, dans un système embarqué, un processus ne peut être interrompu brusquement sans prendre un certain nombre de précautions telles que la libération de zones mémoires allouées ou la réinitilisation ou l'enregistrement de données. A l'heure actuel, le génie logiciel n'offre pas de méthodes générales propres.

Nous avons donc un codel par phase de traitement. Cette décomposition des traitements en différentes phases offre en outre:

- Une programmation structurée qui aide la conception du programme en explicitant les phases de traitement et évite l'usage de variables d'état.
- La connaissance, durant l'exécution, du stade du traitement de l'activité, particulièrement utile durant l'intégration et les mises au point (débogue) souvent nécessaires.
- Une spécification explicite par le programmeur des points d'interruption possibles du traitement: les transitions d'une phase à l'autre.
- La réutilisation de portions de code communes, les codels, par différentes activités. Par exemple les phases d'initialisation et de terminaison d'exécution de trajectoires sont identiques

indépendamment du type de la trajectoire. Une même segmentation d'image peut être utilisée dans des procédures distinctes de localisation ou d'identification d'obstacles.

Ainsi une activité consiste en une succession de traitements prédéfinis, mais dont le séquencement est déterminé dynamiquement selon l'information retournée à l'issu de la phase précédente du traitement. L'enchaînement est donc dirigé par des critères fonctionnels, indépendamment d'aspects de contrôle de l'activité présentés dans la section précédente. Ce séquencement peut également être représenté par un graphe, nommé *graphe fonctionnel*. Notons que la fonction exécutée durant l'état **INTER**, qui est lui-même ininterruptible, est également un codel.

2.3.2 Graphe fonctionnel d'une activité

Le graphe fonctionnel représente l'ensemble des phases de traitement par lesquelles une activité *peut* transiter. A défaut d'information fonctionnelle préalable au niveau du contrôleur, la première étape d'exécution invoquée sur la transition **exec/-** est unique et impérative. Cette phase, marquée **exec.0** sur la figure, va initialiser le traitement. Par la suite, l'activité va transiter par un certain nombre de sous-états fonctionnels **exec.i** de l'état **EXEC** représentés par la figure 2.3.

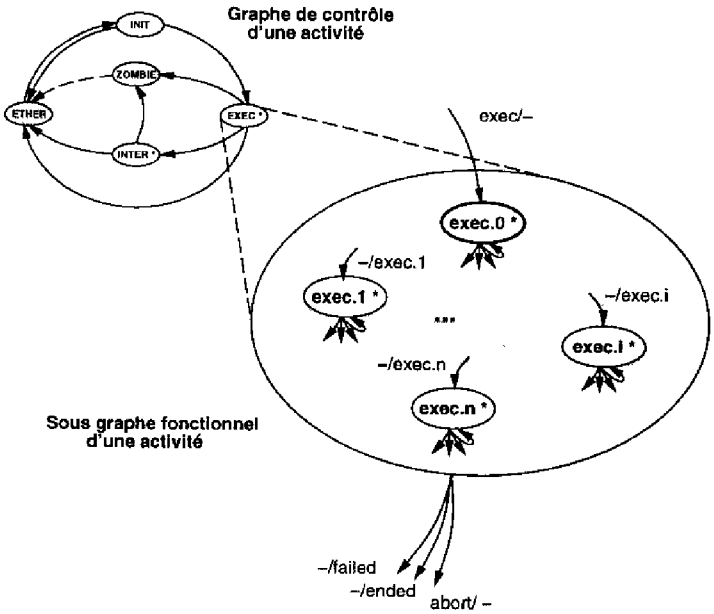


FIG. 2.3 – *Graphe d'état fonctionnel d'une activité.*

Dans cette représentation, **EXEC** est un macro-état qui encapsule des sous-états. Si une activité est dans l'état **EXEC** alors elle réside dans un et un seul des ses sous-états **exec.i**. L'observation d'un des trois événements **abort/-**, **-/failed** ou **-/ended** impose la sortie de l'état **EXEC** et par conséquent du sous-état actif. L'état **exec.0** est le point d'entrée du graphe fonctionnel de l'activité.

Le nombre d'états fonctionnels possibles ainsi que les fonctions de traitement qui leur sont associées sont fixés pour chaque activité lors sa définition. Tous ne seront pas nécessairement invoqués pendant la durée de vie de l'activité.

L'activité transite dans un état **exec.j** ($j \in [0..n]$) suite à un événement **-/exec.j** spécifié avec la fin de l'état précédent **exec.i**.

L'appel itératif d'une fonction à caractère répétitif (en particulier périodique) dont l'exécution est représentée par l'état **exec.i** s'obtient en générant à nouveau l'événement **-/exec.i** à l'issu du traitement **exec.i**.

A chacun de ces états correspond un codel.

2.3.3 Une instance du graphe fonctionnel

Pour de nombreuses applications on peut dégager quatre phases dans le déroulement d'une activité: une phase d'initialisation, le corps principal de l'exécution, une phase de terminaison et éventuellement une procédure spécifique de terminaison en cas d'échec de la boucle principale. C'est pourquoi nous proposons en standard, dans le cadre de la génération automatique des modules, quatre états d'exécution auxquels nous avons attribué des noms plus explicites que **exec.i** qui sont respectivement: **start**, **exec**, **end** et **fail** (figure 2.4). Les transitions prennent également le nom de l'état vers lequel elles amènent.

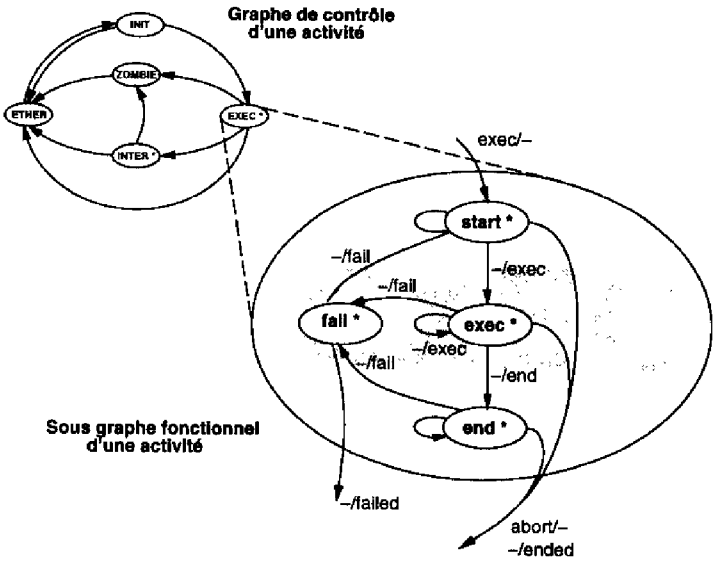


FIG. 2.4 – *Une instance particulière du graphe d'état.*

2.3.4 Séquencement des activités

On distingue deux aspects dans le séquencement des codels d'une activité: le séquencement logique qui définit quel est le prochain codel à exécuter, et le séquencement temporel qui précise l'instant d'invocation de ce codel.

2.3.4.1 Séquencement logique des activités

L'exécution d'une activité est initialisée par événement de contrôle externe **exec/-** qui amène dans l'état **exec.0**. L'activité transite ensuite par une succession d'états qui sont séquencés par des événements internes **-/exec.i**. Deux de ces événements produits par l'activité, **-/ended** et **-/failed** marquent la fin du traitement, le contrôleur reprend alors la main sur l'activité. Ce séquencement, dirigé par l'activité elle-même peut cependant être interrompu

à tout instant par le contrôleur: l'événement d'interruption **abort/-** est préemptif sur tout autre événement. Autrement dit, dès la fin de l'exécution du codel, l'activité va transiter dans l'état **INTER**.

2.3.4.2 Séquencement temporel des activités

Le séquencement temporel des activités durant l'exécution dépend du caractère périodique ou apériodique du traitement, et d'interactions avec d'autres modules. Ainsi l'événement interne issu de la fin d'une phase d'exécution peut se traduire par un signal de réactivation de l'activité qui peut être périodique (P), immédiat (I) ou différé (D). Cette information est transmise par le codel au séquenceur de l'activité (présenté à la section suivante) avec le type de l'événement interne: **-/exec.i(S)**, avec **S** $\in$ **{P, I, D}**.

▷ **Les traitements périodiques** sont séquencées par une horloge. Chaque top d'horloge active la phase suivante d'exécution selon la période qui a été spécifiée. Entre la fin de l'exécution d'un codel et le top suivant de l'horloge, l'activité est en sommeil. Dans l'exemple de la figure 2.5 page ci-contre, l'activité 1 après la phase d'initialisation **exec.0** exécute périodiquement la phase **exec.1**.

▷ **Les traitements immédiats** enchaînent directement les phases de traitements successives tant qu'aucune interruption n'intervient. Dans l'exemple de la figure 2.5 page suivante, l'activité 2 enchaîne les phases **exec.0** et **exec.1**.

▷ **Les traitements différés** permettent d'attendre de façon passive l'occurrence d'un événement externe pour passer à l'exécution suivante. Cet événement est soit issu du contrôleur, soit la réplique à une requête émise par l'activité. Cet événement d'éveil peut également être émis par une autre activité du module.

En effet, lorsqu'une activité requiert un service d'un autre module, *elle ne doit en aucun cas attendre la réplique de façon bloquante* dans un codel car l'activité serait alors paralysée et ininterruptible. Deux possibilités sont alors envisageables: a) par scrutation: l'activité regarde périodiquement si la réplique est disponible, b) par interruption: le traitement est différé jusqu'à l'occurrence d'un événement externe. Dans l'exemple de la figure 2.5 page ci-contre, l'activité 2 diffère l'exécution de la phase **exec.2** jusqu'à l'occurrence de la réplique d'un service requis durant la phase **exec.1**.

Les chronogrammes de la figure 2.5 page suivante montrent en outre un cas de conflit entre les activités 1 et 2. Remarque: si l'activité 2 avait été requise légèrement plus tôt et que l'événement **abort/-** émis auprès de l'activité 1 se fut produit entre le temps $n+2$ et l'événement interne **-/exec.1(P)**, alors les chronogrammes seraient inchangés, seul l'état **INIT** de l'activité 2 aurait démarré un peu plus tôt.

2.3.5 Ressource processeur et priorités

Les activités utilisent deux ressources qu'elles doivent se partager: le temps CPU que laisse apparaître le chronogramme de la figure 2.5 page ci-contre, et la mémoire. L'absence de maîtrise de ces ressources peut conduire à des dysfonctionnements temporaires ou définitifs inacceptables pour un système autonome. Il est cependant très difficile de prédire les besoins qui dépendent des capacités de la machine hôte. C'est pourquoi nous préconisons *des moyens*

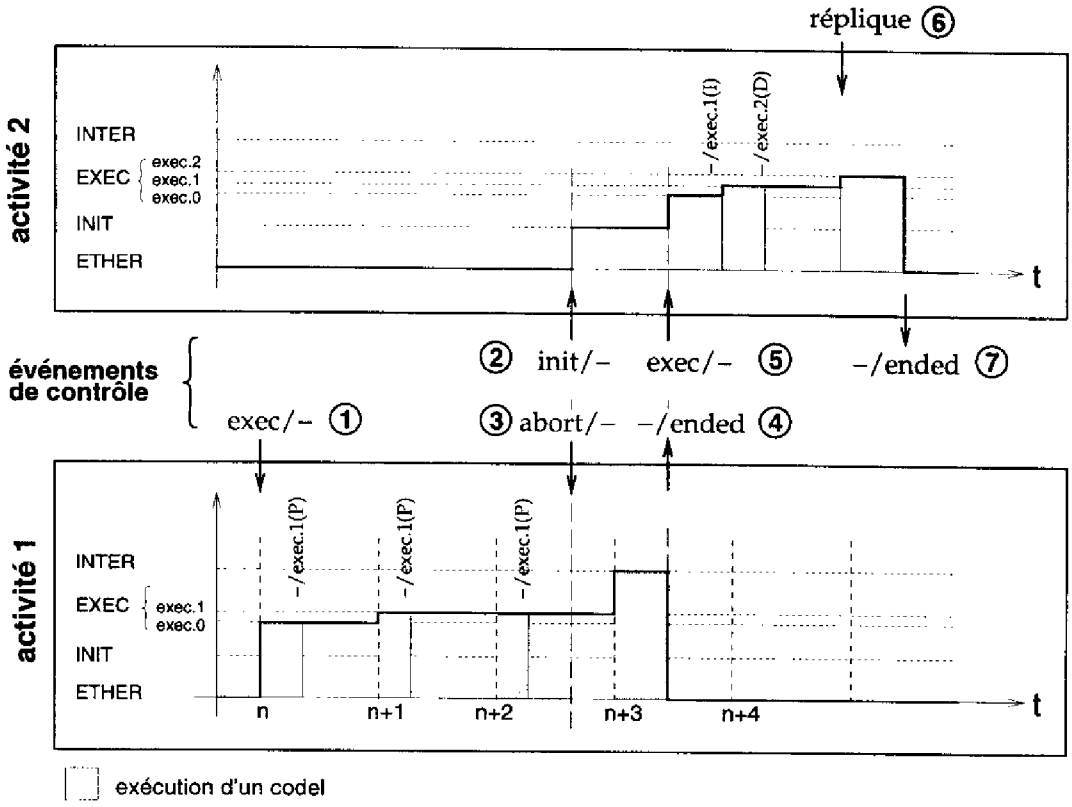


FIG. 2.5 – **Exemple de chronogramme d'activités.** *L'activité périodique 1 a démarré immédiatement. L'activité 2, en conflit avec la première, est maintenue dans l'état* **INIT** *jusqu'à la terminaison effective de l'activité 1. L'activité 2 asynchrone exécute les phases* **exec.0** *puis* **exec.1** *durant lequel elle a requis un service externe. Elle se suspend dans l'attente de la réplique qui l'amènera dans sa phase de terminaison* **exec.2**.

de mesure et de contrôle a posteriori pour estimer les besoins durant les phases de mise au point.

▷ **La ressource mémoire** est particulièrement cruciale car une famine se traduit généralement par un blocage de l'ensemble des modules hébergés par le processeur. Une façon de résoudre définitivement ce problème, serait d'interdire l'utilisation de mémoire dynamique. Cette restriction étant beaucoup trop contraignante, elle n'est appliquée que pour les fonctions de base (communication, asservissement, ...) dont l'échec à cause d'un manque de mémoire serait difficilement acceptable. Les allocations au niveau des codels devront être estimés par le développeur et validés par des tests. Pour les codels qui déroulent des algorithmes structurellement invariants cette estimation est triviale. Les systèmes d'exploitation offrent des outils efficaces pour mesurer la consommation mémoire de chaque processus.

▷ **La ressource en temps** est plus ou moins sensible selon la dynamique du processus contrôlé, les points névralgiques étant les activités d'asservissement et de surveillance. Cependant ces activités sont généralement structurellement invariantes: la durée d'exécution du codel étant constante, leur consommation en temps à chaque période est aisément mesurable.

Il faudra s'assurer que la somme des rapports entre la durée d'exécution d_i et la période T_i de chaque activité i ($\sum \frac{d_i}{T_i} < 1$) soit inférieure à 1 afin de conserver du temps pour les autres traitements (non périodiques). Pour toutes les activités, les priorités d'accès à la CPU devront être proportionnelles au temps de réponse souhaité et donc, pour les activités périodiques, inversement proportionnelles à la période. Les activités non périodiques auront des priorités inférieures aux activités périodiques.

S'il s'avère que les activités de priorité supérieure laissent trop peu de temps CPU aux autres activités qui, de ce fait, ne peuvent respecter leur période imposée ou ont des performances en temps de traitement qui ne sont pas satisfaisantes, il faudra alors les répartir sur d'autres processeurs. Des outils permettent de détecter et d'analyser ce type de situations: les systèmes d'exploitation offrent des fonctions pour mesurer la charge d'un processeur et la répartition de cette charge pour chaque processus, et surtout, les modules mesurent et exportent les temps de calcul des activités dans les posters de contrôle; les débordements par rapport aux estimations sont ainsi signalés et peuvent être surveillé automatiquement. Enfin, pour procéder à une analyse plus fine, nous proposons également des outils de visualisation des chronogrammes des activités (voir les figures 1.11 page 93 et 1.12 page 93 de la partie III).

Enfin, l'ordre de déclenchement d'activités périodiques de même période peut être maîtrisé en imposant des décalages d'activation par rapport à une horloge absolue.

2.4 Les interactions à l'intérieur d'un module

Les données partagées entre les activités d'un module, ou entre ces activités et le contrôleur, sont stockées dans les SDIs. L'exclusion mutuelle d'accès aux SDIs est assurée par des sémaphores. La SDI/c, qui regroupe l'ensemble des informations ayant trait au contrôle des activités et à l'état global du module (événements, états des activités, temps d'exécution, bilans, *etc.*) a une structure identique pour tous les modules. La SDI/f contient les données fonctionnelles qui dépendent des services offerts. Elle comporte les paramètres des requêtes, les résultats des traitements et les données échangées entre les activités.

Le flux de données au sein d'un module est représenté par la figure 2.6 page suivante.

Les paramètres des requêtes sont validés par des codels spécifiques (ou *fonction de contrôle*), exécutés par le contrôleur, qui ont accès à l'état courant de l'exécution grâce aux SDIs.

2.5 Structure logicielle d'un module

D'un point de vue informatique, la structure d'un module découle de sa description en termes de comportement et de fonctionnalité. Ainsi que le montre la figure 2.7 page ci-contre le module se compose:

- d'une tâche de contrôle: le contrôleur,
- de plusieurs tâches d'exécution qui déroulent et séquencent les activités,
- de deux bases de données: les SDIs,
- des boîtes aux lettres qui réceptionnent répliques et requêtes,
- de zones mémoires partagées multi-processeurs: les posters,
- des bibliothèques d'accès aux boîtes aux lettres et aux posters.

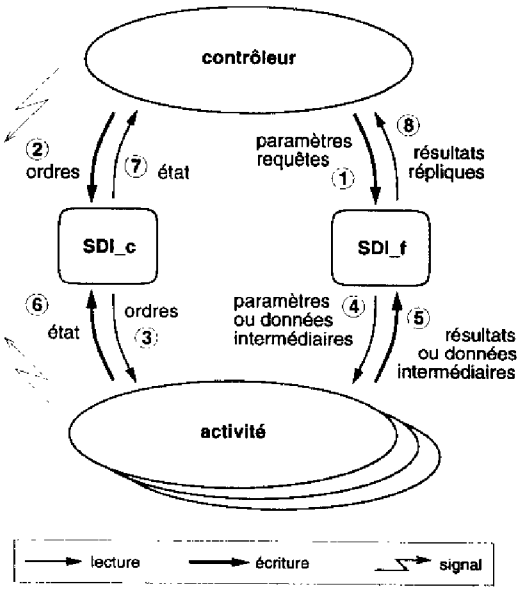
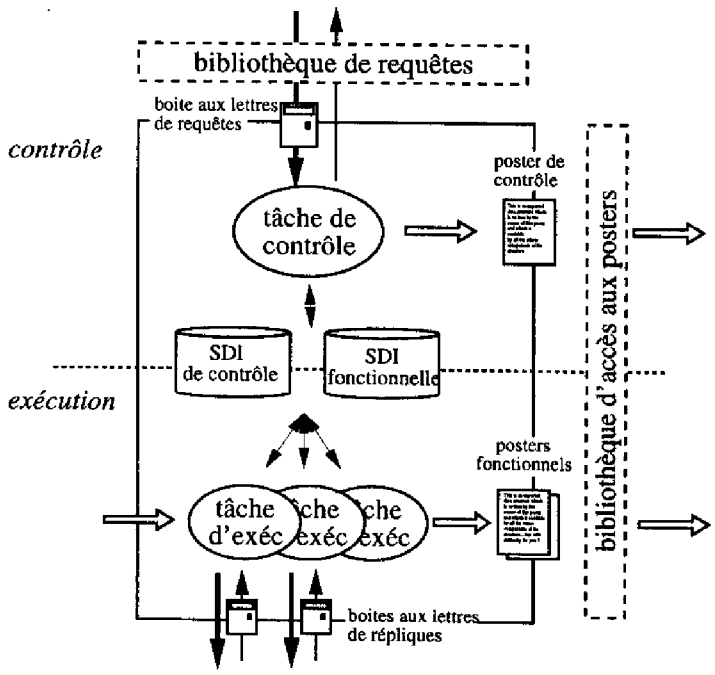


FIG. 2.6 – *Interactions intra-modulaires:*

① *Enregistrement des paramètres des requêtes après validation*
② *Ordre d'exécution ou d'interruption + signal.*
③ *Réception des ordres.*
④ *Lecture de paramètres.*
⑤ *Enregistrement de données ou résultats intermédiaires.*
⑥ *Mise à jour de l'état courant et événements de terminaison + signal.*
⑦ *Enregistrement du nouvel état et du bilan.*
⑧ *Lecture des résultats à retourner.*

FIG. 2.7 – *Structure logicielle d'un module.*



2.5.1 Le contrôleur de module

Le contrôleur a un rôle pivot, il doit réagir aux requêtes des clients et aux changements d'état des activités. Il est implémenté par une tâche asynchrone qui se réveille à l'occurrence d'événements, internes ou externes, et procède aux lancements/interruptions d'activités et à l'émission des répliques. Outre ces opérations de gestion de durée négligeable, la tâche de contrôle n'effectue aucun traitement, ce qui la rend disponible et donc réactive aux requêtes ou aux changements d'états.

L'algorithme de la tâche de contrôle, présenté ci-dessous, consiste, à l'occurrence d'un événement, à traiter les transitions internes puis à réceptionner les requêtes. Il en déduit les activités à démarrer. Les algorithmes de ces trois fonctions sont également développés.

```
TÂCHE DE CONTRÔLE ()

Initialisation;
boucle
    | Attente événement;
    | Traiter les événements internes();
    | Traiter les événements externes();
    | Démarrer les activités prêtes();
    | Mettre à jour le poster de contrôle;
```

```
Traiter les événements internes ()

pour chaque (événement interne)
    | Mettre à jour les graphes d'état;
    | Envoyer les répliques finales des activités terminées;
```

```
Traiter les événements externes ()

pour chaque (requête)
    | status = Exécuter la fonction de contrôle;
    | si (status == OK)
    |     | Mettre à jour la SDI/f;
    |     | Interrompre les activités incompatibles (abort/-);
    |     | si (requête d'exécution)
    |     |     | Enregistrer l'activité (init/-);
    | sinon
    |     | si ( requête d'exécution)
    |     |     | Envoyer la réplique finale avec bilan d'erreur;
    | si (requête de contrôle)
    |     | Envoyer la réplique finale au client;
```

```
Démarrer les activités prêtes ()

pour chaque (activité en attente (INIT))
    | si (pas d'activité incompatible)
    |     | Démarrer l'activité (exec/-) ;
```

2.5.2 Le niveau d'exécution du module

Les activités sont exécutées par des tâches dénommées *tâches d'exécution* qui fournissent le contexte d'exécution des codels. Une tâche d'exécution peut être chargée de plusieurs fonctions de traitement et donc d'activités, en particulier si ces activités sont plusieurs instances

d'un même traitement (par exemple des surveillances). Les divers traitements devront cependant présenter des dynamiques similaires (même période ou même ordre de temps d'exécution) et des priorités équivalentes car elles héritent ces caractéristiques de la tâche d'exécution. Dans la mesure où ces conditions sont respectées, la répartition des traitements entre les tâches reste à la discrétion du programmeur. Il est néanmoins conseillé de limiter le nombre de tâches afin de réduire les temps de changement de contexte.

L'algorithme des tâches d'exécution, très simple, est décrit ci-dessous. Une tâche d'exécution réagit aux signaux suivants:

- aux tops d'horloge,
- aux événements de contrôle,
- à la réception d'une réplique,
- aux événements internes de changement d'état.

```
TÂCHE D'EXÉCUTION ()

Initialisation;
boucle
    | Attente signal;
    | Exécuter les activités permanentes;
    | pour chaque (activité)
    |     | selon (type transition)
    |     |     | cas exec/-    : Exécuter le codel exec.0;
    |     |     | cas abort/-   : Exécuter le codel inter;
    |     |     | cas -/exec.i  : Exécuter le codel exec.i;
    |     | Enregistrer la transition suivante;
    | Mettre à jour les posters;
```

2.5.3 Les structures de données internes (SDI)

Les structures de données internes sont des zones de mémoire locales à un processeur et protégées par un sémaphore. Comme nous l'avons vu, un module dispose de deux SDI distinctes:

▷ **La SDI de contrôle** est standard, elle se compose de trois éléments:

- une sous-structure qui décrit la tâche de contrôle (identificateur, status, bilan pour identifier l'erreur en cas de défaillance, nombre d'activités),
- un tableau de sous-structures qui décrivent les tâches d'exécution (identificateurs, status, bilan, activités en cours, périodicité, ...),
- un tableau de sous-structures qui décrivent les activités en cours (identificateur, type de requête, état et sous-état, bilan d'exécution, adresses des paramètres dans la SDI/f, ...).

▷ **La SDI fonctionnelle** est spécifique pour chaque module. Elle est décrite par le développeur du module. Elle regroupe en particulier les zones de stockage des paramètres des requêtes et des répliques. On conserve ainsi une image des dernières données transférées ce qui s'avère très utile pendant des opérations de débogue. Les zones de stockage qui concernent des fonctions qui peuvent avoir plusieurs instances actives simultanément sont dupliqués.

2.5.4 Les bibliothèques de communication inter-modules

Ne disposant pas sur le marché de bibliothèques de communication qui soient en adéquation avec nos besoins, deux bibliothèques ont été spécifiées et conçues au LAAS: `posterLib` et `csLib`. Elles implémentent les deux protocoles que nous présentons succinctement ici, sur les systèmes d'exploitation UNIX et VxWorks et permettent également des transferts entre ces deux domaines. Pour plus d'informations, on pourra se référer aux thèses de R. Ferraz de Camargo [Ferraz De Camargo 91] et V. Perebaskine [Perebaskine 92].

▷ **La communication client/serveur: `csLib`.** La bibliothèque de communication `csLib` implémente un protocole client/serveur de haut niveau. Les fonctions offertes par cette bibliothèque sont l'émission et la réception asynchrone de requêtes et de répliques. Une requête fait directement référence à un service, c'est à dire à une fonction spécifique au niveau d'un serveur. Ces messages typés sont réceptionnés dans des zones mémoire tampon, les boîtes aux lettres, jusqu'à ce qu'ils soient effectivement lus par le destinataire.

Une bibliothèque de gestion des requêtes de chaque module est construite à partir de `csLib`: A chaque service correspond un jeu de fonctions d'émission de la requête et de réception des répliques intermédiaire et finale.

▷ **La communication par posters: `posterLib`.** La bibliothèque `posterLib` permet de créer, de mettre à jour ou d'accéder en lecture les posters. Chaque poster est la propriété d'une tâche (dans le cadre des modules il s'agit d'une tâche de contrôle ou d'une tâche d'exécution) qui seule peut en modifier le contenu.

Une bibliothèque d'accès aux posters de chaque module est construite à partir de `posterLib`: a chaque poster correspond un jeu de fonctions de lecture de la structure et des sous-structures qui la compose.

Chapitre 3

Description formelle et génération automatique

Les modules ont une structure, un fonctionnement et des interactions bien définis dont la standardisation nous a permis d'élaborer un "langage" [1] déclaratif permettant de décrire de façon formelle chaque module. La génération automatique des modules découle naturellement de cette description formelle. A cette fin nous avons développé un générateur de modules nommé G^{en}$_{o}$M pour "Generator of Modules" qui produit un module complet incluant les bibliothèques d'interface et un programme interactif de test.

Nous allons présenter successivement le langage de spécification des modules et le générateur de modules. Puis nous proposerons une méthodologie de développement, d'intégration et de test d'un module.

3.1 Description formelle

La description formelle permet de spécifier entièrement un module depuis les services offerts: les requêtes, les posters et les types de données manipulées, jusqu'à son fonctionnement interne: le séquencement des activités, les codels associés, les compatibilités, les bilans, *etc.*. Cette description comporte cinq parties que nous aborderons successivement:

1. La déclaration globale du module
2. La déclaration de la SDI fonctionnelle
3. La déclaration des requêtes
4. La déclaration des posters
5. La déclaration des tâches d'exécution

La syntaxe employée est proche du langage C. Les déclarations des types de données sont d'ailleurs exprimées dans ce langage, ce qui offre l'avantage de pouvoir inclure directement dans la description formelle des fichiers source C de déclaration de type de structure de données.

Hormis la déclaration de la SDI/f qui est un type C, les déclarations ont toutes la même structure: une en-tête composée d'un mot-clé pour caractériser l'objet: `module`, `request`,

1. Le terme langage est peut-être improprement employé dans la mesure où il ne s'agit pas d'un langage informatique général. A défaut de terme mieux adapté, nous le conserverons cependant. Il devra être interprété dans le sens restreint de *langage de spécification*.

`poster` ou `exec_task`, et d'un nom pour l'identifier, suivie d'une liste d'attributs entre accolades {}. Chaque attribut se compose d'un mot-clé suivi des paramètres du développeur.

Le nom attribué à chaque objet joue un rôle très important car il permettra à G^en^oM de constituer les noms des composants générés: les fonctions des bibliothèques de requêtes et de posters, les SDIs, les messages d'erreur, *etc.* La composition des noms suit une règle simple et systématique: ils sont la concaténation du nom du module, du nom de l'objet concerné, et du terme qui caractérise l'élément généré. Ainsi la fonction de lecture d'un poster nommé `robot` du module `loco` sera: `locoRobotPosterRead`[2].

Dans la présentation qui suit, le signe '|' indique un choix exclusif de paramètre alors que ';', ':' et '::' sont des séparateurs du langage. Dans une liste, signalée par '...', les paramètres devront être séparés par des virgules. Les mots-clés du langage sont écrits en style `courrier` et les autres termes en *italique*. On trouvera en annexe A la grammaire complète du langage.

Avertissement: G^en^oM est un produit quotidiennement employé sur nos robots. Son langage de spécification peut difficilement évoluer sans perturber les utilisateurs, et les changements souhaitables, qui ne remettent pas en cause les principes énoncés au chapitre précédent, ne seront apportés que lors d'une version ultérieure. Nous avons fait le choix d'en présenter ici la version courante en précisant les évolutions, relativement limitées, qui sont envisagées.

3.1.1 Déclaration globale du module

La déclaration globale du module permet d'attribuer un nom et un numéro d'identification au module et d'indiquer le type de la SDI/f.

```
module identificateur {
    number:          expression;
    internal_data:   nom-typedef;
    params:          identificateur, ... ;
};
```

Le nom du module, indiqué après le mot-clé `module`, servira de préfixe à l'ensemble des fichiers, bibliothèques et fonctions d'interaction (requêtes, posters) du module.

- `number`: Numéro d'identification qui permet de coder les bilans d'exécution de façon unique au sein de la couche fonctionnelle.
- `internal_data`: Nom du type de la SDI/f décrite au paragraphe suivant.
- `params`: Liste de noms de variables. Elles sont initialisées selon les valeurs passées en paramètre de la fonction de démarrage du module et accédées depuis la SDI/c.

3.1.2 Déclaration de la SDI fonctionnelle

La SDI/f est la base de données par laquelle transitent toutes les données fonctionnelles des activités. Sa structuration doit être prédéfinie. Elle est décrite selon le formalisme de définition de structures des données du langage C (`typedef struct ...`).

2. Les noms sont établis selon la règle de composition suivante: les noms des variables et fonctions commencent par une minuscule et les mots qui les composent sont séparés par une majuscule.

Les paramètres des requêtes et des répliques (et donc des fonctions de traitement), les données exportées dans les posters et éventuellement les données partagées entre les activités du module doivent impérativement disposer d'une zone mémoire dans la SDI/f sous la forme d'une sous-structure. Ces sous-structures sont définies dans des fichiers de déclaration que l'on inclut directement grâce à la directive `#include`.

Si le module accède à des services ou des données exportées d'un autre module il devra alors manipuler des structures déjà définies par cet autre module[3]. Les fichiers d'en-tête de déclaration de ces structures communes seront également inclus par la directive `#include` mais celle-ci devra être intégrée dans une déclaration "`import from` *identificateur*" où *identificateur* est le nom du module serveur.

Un exemple très général de déclaration de SDI/f est donné ci-dessous:

```
import from identificateur {          /* Inclusion des structures prédéfinies */
    #include "decla-struct-serveur.h"
    ...
}
...
#include "decla-struct-module.h"      /* Inclusion des structures du module */
...
typedef struct identificateur {       /* Définition de la SDI/f */
    ...(c-like declaration)
}
```

3.1.3 Déclaration des requêtes

3.1.3.1 Les requêtes de contrôle

Chaque requête a un nom indiqué dans l'en-tête de la déclaration. Le seul attribut obligatoire est le champ `type` qui distingue les requêtes de contrôle des requêtes d'exécution. Dans le cas présent son attribut est le mot-clé `control`. Les autres attributs sont optionnels.

```
request identificateur {
    type:                 control;
    input:                variable-c::référence-sdi;
    output:               variable-c::référence-sdi;
    c_control_func:       identificateur;
    incompatible_with:    none | all | identificateur,...;
    fail_msg:             identificateur,...;
}
```

- `input`: Argument (optionnel) de la requête.
  Paramètres: Nom attribué au paramètre suivi de son adresse dans la SDI/f.
- `output`: *Idem* pour le résultat éventuellement retourné dans la réplique.

3. Une règle élémentaire pour maintenir la cohérence des interactions est l'unicité de déclaration des structures partagées.

- `c_control_func`: Nom du codel (optionnel), invoquée par le contrôleur, qui permet de vérifier l'applicabilité de la requête (en particulier la validité de ses paramètres).
- `incompatible_with`: Spécifie les activités incompatibles à interrompre.
  Paramètres: `none` (valeur par défaut) ou `all` ou liste de requêtes d'exécution.
- `fail_msg`: Liste des messages d'erreur qui peuvent être retournés par la fonction de contrôle et transmis dans le bilan de la réplique à la requête.

3.1.3.2 Les requêtes d'exécution

Les requêtes d'exécution se distinguent des requêtes de contrôle par le mot-clé `exec` dans le champ `type`. On retrouve ici les attributs des requêtes de contrôle auxquels vont s'ajouter des attributs relatifs à l'activité. Par extension, le nom attribué à l'activité est celui de la requête correspondante.

```
request identificateur {
    type:                exec;
    input:               variable-c::référence-sdi;
    output:              variable-c::référence-sdi;
    c_control_func:      identificateur;
    incompatible_with:   none | all | identificateur,...;
    fail_msg:            identificateur,...;
    c_exec_func:         identificateur;
    c_exec_func_start:   identificateur;
    c_exec_func_end:     identificateur;
    c_exec_func_fail:    identificateur;
    c_exec_func_inter:   identificateur;
    exec_task:           identificateur;
    resources:           identificateur,...;
    activity_type:       filter|server|servo_process|surveillance;
}
```

Les champs communs aux requêtes de contrôle ne sont pas redécrits ici. Parmi les champs supplémentaires, seul `exec_task` est obligatoire.

- `c_exec_func`: Codel associé au sous-état **exec**.
- `c_exec_func_start`: Codel associé au sous-état **start**.
- `c_exec_func_end`: Codel associé au sous-état **end**.
- `c_exec_func_fail`: Codel associé au sous-état **fail**.
- `c_exec_func_inter`: Codel associé au sous-état **inter**.
- `exec_task`: Tâche d'exécution chargée de l'exécution de l'activité.
- `activity_type`: Type de l'activité.
  Paramètre: `filter` ou `server` ou `servo_process` ou `surveillance`.
- `resources`: Liste des serveurs ou des ressources utilisés par l'activité.

Remarques: Les codels sont tous optionnels. Or nous avons vu que le point d'entrée du graphe d'exécution d'une activité est l'état **start**. Si celui-ci n'est pas défini, alors c'est la phase **exec** qui sera invoquée, et par défaut la phase **end**. Il est évident que si aucun n'est

défini l'intérêt de la requête d'exécution est douteuse (retour immédiat à l'état **ETHER**).

Soulignons ici l'importance du champ `incompatible_with`. Il permet en particulier d'indiquer si l'activité est compatible avec elle-même, auquel cas plusieurs de ces activités (autant que de requêtes) pourront s'exécuter en parallèle. Une conséquence, transparente pour le développeur, est la duplication dans la SDI/f des sous-structures qui réceptionnent les paramètres de la requête et de la réplique afin d'éviter tout conflit lors d'instances multiples d'activités.

Evolutions: Actuellement on ne peut donc définir que quatre phases de traitement. La version à venir sera généralisée à un nombre quelconque de phases d'exécution **exec.i**. La phase initiale, alors impérative, sera déclarée selon la syntaxe actuelle (`c_exec_func_start`). La définition des autres états comportera deux paramètres: le premier désigne le nom de l'état et le second celui du codel associé:

```
c_exec_func_start:   identificateur;
c_exec_func:         identificateur :: identificateur;
```

3.1.3.3 Les codels

Les codels sont des fonctions C dont le code est écrit par le concepteur du module, et qui, rappelons-le, ne peuvent pas être interrompus par le contrôleur[4]. Pour pouvoir intégrer ces codels au module, ceux-ci doivent respecter le prototypage de G^{en}oM:

- Les codels liés à la fonction de contrôle (`c_control_func`) prennent en paramètre: le paramètre de la requête (en fait son adresse dans la SDI/f) et le bilan d'exécution (son adresse dans la SDI/c) auquel ils doivent attribuer une valeur particulière en cas de problème (voir §3.1.3.4). Ils retournent un booléen qui, s'il est faux, termine immédiatement le traitement, sans enregistrer les paramètres de la requête dans la SDI/f.
- Les codels liés à l'exécution d'une activité (`c_exec_func_...`) ont les mêmes paramètres, auxquels s'ajoute l'adresse dans la SDI/f de la structure de résultat (si la réplique inclut un résultat). Ils retournent en fin de traitement le type d'événement interne qui sélectionnera l'étape suivante (**exec.i**, **ended** ou **failed**).
- Les codels liés à l'exécution d'une activité permanente (`c_func`, déclaré avec la tâche d'exécution - voir §3.1.5) ont pour unique paramètre le bilan et retournent un booléen.
- Les codels d'initialisation (`c_init_func`, déclaré avec la tâche d'exécution - voir §3.1.5) ont pour unique paramètre le bilan et retournent un booléen.

Pendant leur déroulement les codels peuvent accéder directement aux SDIs qui sont protégées par exclusion mutuelle.

3.1.3.4 Les bilans

Chaque bilan déclaré dans les champs `fail_msg` est identifié par un code unique qui est associé à un message. Le code et le message sont générés par G^{en}oM en respectant les

4. Cela n'empêche pas le système d'exploitation de procéder à des changements de contexte lorsque des tâches plus prioritaires doivent s'exécuter.

conventions de codage VxWorks[5]. Cela permet d'identifier de façon discriminante l'origine du dysfonctionnement et le dysfonctionnement lui-même, y compris s'il a pour origine une erreur système. Par exemple, l'information `EMERGENCY_STOP` que pourrait générer une activité d'un module `LOCO` se traduira par le message: `S_locoCmdTask_EMERGENCY_STOP`.

Durant l'exécution d'une activité, des dysfonctionnements indépendants des codels peuvent se produire (erreur système, interruption, *etc.*) donnant lieu à des bilans spécifiques sans référence au module (*e.g.* `S_memLib_NOT_ENOUGH_MEMORY`). Pour les remonter tout en pointant sur le module concerné, un certain nombre de bilans définis et gérés par G^{en}$_{o}$M s'ajoutent aux bilans déclarés par le développeur, tels que `S_locoCntrlTask_SYSTEM_ERROR` ou `S_locoCntrlTask_ACTIVITY_INTERRUPTED`. Les bilans originaux, qui précisent la cause du dysfonctionnement, sont maintenus accessibles dans le poster de contrôle.

3.1.4 Déclaration des posters d'exécution

Les posters d'exécution exportent des données produites par les activités. Ils sont mis à jour, soit automatiquement lors de l'invocation de l'activité, soit sur demande explicite d'un codel. Ces deux cas se distinguent par les mots-clés `auto` et `user` dans le champ `update`. De même que les SDIs, les posters sont structurés selon le formalisme des structures du langage C. Les fonctions générées par G^{en}$_{o}$M permettent d'accéder directement à n'importe quelle sous-structure du poster ou à la structure complète.

Mise à jour automatique: Les sous-structures qui composent le poster sont listées dans le champ `data`. Les couples d'attributs pour chacune d'entre elles sont composés d'un nom, qui permettra de la référencer, et de son adresse dans la SDI/f. Les paramètres du champ `activity` identifient l'activité et la transition sur laquelle le poster doit être mis à jour.

```
poster identificateur {
    update:     auto;
    data:       identificateur::référence-sdi,...;
    activity:   identificateur::identificateur;
};
```

Mise à jour sur demande: Les posters mis à jour "manuellement" sont beaucoup plus souples à utiliser. En contrepartie ils ne peuvent être entièrement gérés par le module. Ce cas est le seul pour lequel l'opérateur devra lui-même écrire des fonctions d'interaction: les fonctions de lecture et d'écriture des posters. Il est bien sûr plus que recommandé de respecter les standards de syntaxe et de dénomination.

La mise à jour du poster devra donc être codée dans les codels eux-mêmes. La possibilité de pouvoir définir la procédure de remplissage du poster est intéressante dans quelques cas précis. En effet, le principe qui consiste à définir les structures des posters sous forme de structures du langage C n'est pas adapté à tous les types de données. Certaines données nécessitent une représentation plus dynamique: par exemple il est difficile de prédéfinir une

5. Le code est la composition du numéro d'identification du module, de la tâche ou de la bibliothèque concernée et du numéro attribué au bilan. Le message est la concaténation du nom de la tâche ou de la bibliothèque concernée et du nom du bilan.

structure qui décrit un ensemble d'obstacles polygonaux dont le nombre de sommets est variable.[6]

```
poster identificateur {
    update:    user;
    type:      nom-typedef;
    activity:  identificateur::identificateur;
};
```

Le champ `type` précise le type de la structure qui compose le poster et plus particulièrement sa taille afin de permettre à G^{en}oM de créer un poster de dimension suffisante.

3.1.5 Déclaration des tâches d'exécution

Les tâches d'exécution ont la particularité de ne pas apparaître explicitement dans la formalisation des modules: elles ne sont que le contexte informatique d'exécution des activités. Leur déclaration permet au développeur de modules d'expliciter la répartition des activités en tâches d'exécution. La déclaration des tâches d'exécution inclut celle des activités permanentes dont ils ont la charge. On remarquera que cette déclaration se présente sous la forme d'un codel unique (`c_func`). En effet ces activités n'ont (après une initialisation) qu'un état possible, elles sont nécessairement exécutées séquentiellement et ne peuvent être interrompues: le codel intègre directement cette séquence d'activités. Exemple: calcul de la position du robot suivi de celui de la commande d'asservissement.

Evolutions: On envisage de se dispenser de la déclaration des tâches d'exécution en reportant les champs relatifs aux activités (marqués d'un astérisque * dans la présentation ci-dessous), dans la déclaration des requêtes d'exécution. Les champs qui concernent l'activité permanente feraient alors l'objet d'une déclaration séparée. Le nombre et les caractéristiques des tâches d'exécution ainsi que la répartition des traitements seraient alors déterminés automatiquement par G^{en}oM.

6. Une description efficace pourrait consister à indiquer successivement le nombre d'obstacles, le nombre de sommets de chaque obstacle, puis les coordonnées de tous les sommets.

```
exec_task identificateur {
    period:                 none | expression;
    delay:                  none | expression;
    priority:               expression;
    stack_size:             expression;
    c_init_func:            identificateur;
    c_func:                 identificateur;
    cs_client_from:         identificateur,... ;
    poster_client_from:     identificateur:: identificateur,... ;
    fail_msg:               identificateur,... ;
    resources:              identificateur,... ;
};
```

- `period*`: Période de la tâche d'exécution, et donc des activités dont elle a la charge, en tics[7] ou none pour les tâches apériodiques.
- `delay*`: Décalage en tics entre les tâches d'exécution périodiques par rapport à une horloge absolue, ou none pour les tâches apériodiques.
- `priority*`: Valeur entre 0 et 255 (de la plus forte à la plus faible) qui exprime les priorités relatives d'accès à la CPU des tâches d'un même processeur (indépendamment du module).
- `stack_size`: Taille de la mémoire allouée à la tâche en nombre d'octets (peut être affinée par des tests).
- `c_init_func`: Fonction d'initialisation invoquée une seule fois qui permet en particulier d'initialiser l'activité permanente et/ou d'attribuer des valeurs par défaut à des données de la SDI/f. *Paramètre:* Nom du codel.
- `c_func`: Nom du codel des activités permanentes.
- `cs_client_from*`: Nom des modules serveurs dont les activités gérées par la tâche d'exécution peuvent potentiellement être client.
- `poster_client_from*`: Les posters dont les activités peuvent potentiellement être lectrices. *Paramètre:* Liste de couples de paramètres: nom du module / nom du poster.
- `fail_msg`: Liste des bilans de l'activité permanente (exporté dans le poster de contrôle).
- `resources`: Liste des éléments du robot logique utilisés par l'activité permanente.

3.2 Le générateur de modules G^{en}$_{o}$M

La description formelle est un premier pas vers la génération automatique des modules qui est franchi grâce au générateur de modules G^{en}$_{o}$M.

Le générateur produit non seulement le module, mais également ses bibliothèques d'interface, qui regroupent les fonctions d'émission de requêtes, de réception de répliques et d'accès aux posters, et un programme interactif de test.

Le code source produit par G^{en}$_{o}$M peut être compilé pour une exécution du module sur une station de travail UNIX ou à bord du robot mobile sous le système temps réel VxWorks.

7. Dans l'implémentation actuelle 1 tic = 5 ms.

Bien évidemment, UNIX n'étant pas un système temps réel on ne peut garantir ni les temps d'exécution ni le séquencement des tâches. Cela reste néanmoins très utile pour effectuer les premiers tests ou pour procéder à des simulations qui n'exigent généralement pas des boucles d'asservissement à haute fréquence.

Les clients des modules (y compris les tâches de test) peuvent s'exécuter indifféremment sur UNIX ou sur VxWorks. Cela permet par exemple de construire des interfaces graphiques conviviales de contrôle du robot par un opérateur grâce aux nombreux outils qu'offre le système UNIX (voir §1.4 de la troisième partie).

3.2.1 Description de G^en^oM

Le générateur, illustré par la figure 3.1, se compose de deux éléments: le squelette d'un module générique et un analyseur syntaxique qui analyse la description formelle et produit le module en instanciant le squelette.

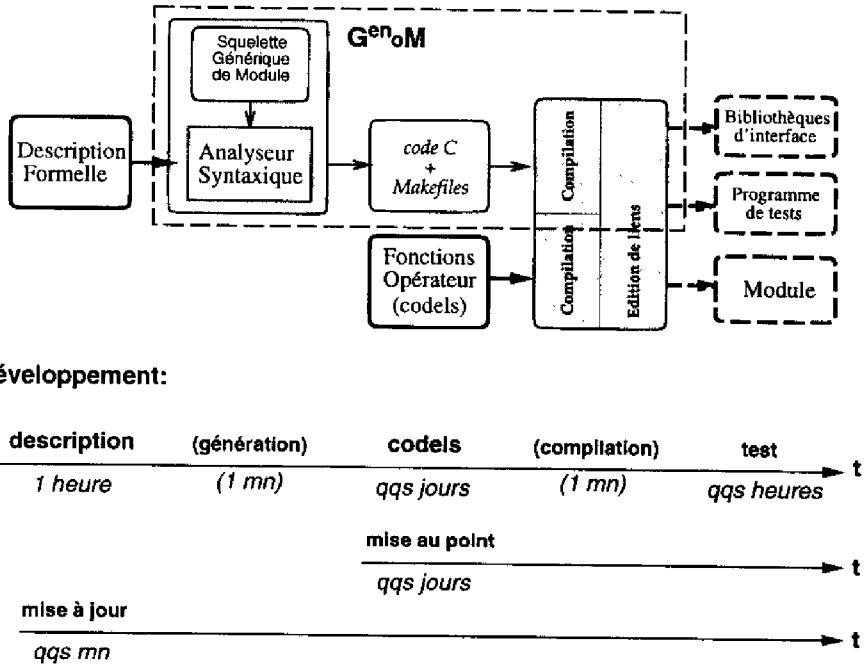


FIG. 3.1 – *Le générateur et le cycle de développement*

Le squelette est un ensemble de programmes C utilisant des mots-clés spécifiques qui peuvent être substitués soit par une simple variable (nom du module, nom d'une requête, période, *etc.*), soit par le code d'une fonction (création d'un poster, déclaration d'un serveur, *etc.*).

la description formelle du module (`fichier.gen`) est d'abord passée au préprocesseur C (`ccp`) qui traite les directives `#include`, `#define`, `#if`, *etc.*. Le résultat est ensuite analysé par l'analyseur syntaxique[8] qui construit une représentation sémantique du module. Celle-ci permet d'instancier les parties variables du code du squelette pour produire les fichiers

8. Développé à l'aide des outils Lex et Yacc.

source du module final, ainsi que les Makefiles nécessaires à la compilation pour produire les fichiers objets correspondant aux différents éléments du module (tâche de contrôle, tâches d'exécution, bibliothèques d'interface, ...). L'édition de lien avec les codels produit enfin le module exécutable.

3.2.2 Le produit de la génération

Les fichiers générés par G^en^oM sont tous préfixés par le nom du module `xxx`. On trouvera en annexe B les fichiers source générés ainsi que les fichiers objet et les exécutables issus de la compilation. Les prototypes des fonctions y sont également décrits.

- **La bibliothèque des requêtes `xxxMsgLib`:** Cette bibliothèque inclut les fonctions d'émission de requêtes et de réception de répliques utilisées par les clients du module.
- **La bibliothèque d'accès aux posters `xxxPosterLib`:** Cette bibliothèque inclut les fonctions de lecture de la structure ou des sous-structures de données des posters de contrôle et des posters d'exécution.
- **Les prototypes des codels et l'accès aux SDIs `xxxHeader.h`:** Ce fichier de déclaration, à inclure dans les bibliothèques des codels, comporte toutes les fonctions nécessaires au développement des codels: prototypage, référence aux SDIs (`xxxDataStrId` et `xxxCntrlStrId`), accès à la SDIc.
- **Liste des bilans `xxxError.h`:** Ce fichier inclut la liste des codes d'erreur et les messages associés. Signalons à cette occasion que bibliothèque (`h2errorLib`) permet de décoder et d'afficher "en clair" un bilan.
- **Le programme de test `xxxEssay`:** Ce programme peut se compiler indifféremment sur UNIX ou VxWorks.
- **Le serveur:** La génération du module produit un fichier source par tâche (`xxxCntrlTask`, `xxxEeeTask`), un fichier incluant la tâche de démarrage du module (`xxxModuleInit`) et deux fichiers de compilation (`Makefiles.vx.xxx` et `Makefile.unix.xxx`). A l'issu de la compilation on dispose d'un fichier objet unique: `xxxModule.o` dont l'édition de liens avec les bibliothèques des codels composera le module complet.

Par défaut, G^en^oM produit des bibliothèques C. Cependant l'analyseur ne fait aucune hypothèse quand au format des fichiers squelettes. Cela permet de produire des bibliothèques PRS, et tout autre langage peut être envisagé.

La génération et la compilation sont bien évidemment exécutées hors ligne et prennent entre une à cinq minutes selon la puissance de la machine. Les fichiers de description formelle comptent généralement 100 à 150 lignes. Les fichiers sources produits comportent environ 2000 lignes pour le serveur, et autant pour les bibliothèques d'interface et pour le programme de test.

3.3 Méthode de développement d'un module

Nous ne proposons pas uniquement une architecture mais également une méthodologie pour intégrer de nouvelles fonctions. La description formelle permet d'orienter le développement du module en dirigeant les questions qu'un développeur doit se poser:

1. Quel est le rôle du module, quels sont ses services principaux?

2. Comment peut-on décomposer les traitements? Ont-ils des étapes itératives ou périodiques?
3. Quand peut-on les interrompre et que faut-il faire en ce cas?
4. Quels sont les paramètres et les ressources nécessaires aux activités?
5. Ont-elles des résultats à retourner? Des données à exporter?
6. Peuvent-elles se dérouler en parallèle? Quelles sont les incompatibilités?
7. Y a-t-il des paramètres que l'on peut fixer par défaut? Est-il intéressant de pouvoir accéder/modifier ces paramètres? Peut-on le faire dynamiquement?
8. Quels problèmes les activités peuvent-elles rencontrer qu'il faudrait signaler?

3.3.1 Cycle de développement d'un module

La méthodologie de conception d'un module comporte quatre étapes que l'on peut réitérer (figure 3.1 page 71):

1. La **description formelle** découle généralement assez naturellement de la spécification. Pour quelqu'un qui connaît le langage, cette description peut s'effectuer en 1 heure environ.
2. La **génération** et la compilation des fichiers résultants prennent quelques minutes.
3. Bien évidemment, le contenu des **codels**, c'est-à-dire les algorithmes, reste à la charge du développeur et leur temps de développement dépend de la complexité du problème. On peut cependant dans un premier temps fournir des codels vides ou presque.
4. Après l'**édition de lien** entre le module et les codels, on obtient le nouveau module, les bibliothèques d'interface et le programme interactif de test. Les premiers **tests** peuvent être menés.

Une première version du module est dès lors disponible dans le système. Par la suite, les codels sont incrémentalement développés et mis au point. Seule une modification dans l'interface du module (adjonction d'une requête, de bilans, changement d'un paramètre) nécessite une nouvelle génération du module. Cette modification est quasi-immédiate et permet une maintenance aisée du module. Notons également que la description formelle du module constitue un document complet (requêtes, posters, types de données) à l'usage des utilisateurs du module.

3.3.2 Procédures de test et de mise au point

Le programme interactif de test se présente sous la forme d'un menu (figure 3.2 page suivante) qui permet d'émettre chacune des requêtes définies (en mode bloquant ou non), de réceptionner les répliques et visualiser les paramètres résultants (bilan d'exécution et l'éventuelle structure résultat), d'interrompre une activité, d'exécuter en parallèle plusieurs activités et de visualiser le poster de contrôle (*i.e.* l'état global du module).

Ce programme est tout simplement un client du module développé qui utilise, comme le feraient d'autres clients, les bibliothèques d'accès au module générées par G^{en}$_{o}$M. On peut exécuter en parallèle plusieurs de ces programmes, établissant ainsi autant de clients.

Le module est généralement testé dans un premier temps en simulation sur UNIX ou sur un rack de développement VxWorks en utilisant les fonctions de simulation du robot logique. Un fois mis au point il pourra être porté directement sur le robot.

```
                5  :  ReedShep
                6  :  Stop
                7  :  Slow
                8  :  MonLength

               66  :  abort activity
               77  :  requests/replies (0 requests on)
               88  :  server state
               99  :  END
pilo0> 2

** EXECUTION REQUEST
** Enter TURN_STR turnStr
dTheta : 1.5
rayon : 2
wMax : 0.8
accel : 1.2
Wait final reply (y/n/a) ? : y
** Request  0 sent
** Activity 0 started
** Final reply S_locoCmdTask_CHOCK

                0  :  AvoidOnOff
                1  :  Move
                2  :  Turn
                3  :  Smooth
                4  :  GlobalSmooth
                5  :  ReedShep
                6  :  Stop
                7  :  Slow
                8  :  MonLength

               66  :  abort activity
               77  :  requests/replies (0 requests on)
               88  :  server state
               99  :  END
pilo0>
```

FIG. 3.2 – **L'interface de test du module d'exécution de trajectoires** PILO. *Les requêtes définies par le module sont émises en composant un numéro de 0 à 8. La requête* Turn *(exécution d'un arc de cercle de 2m de rayon et 1.5rd dans cet exemple) s'est soldée, d'après le bilan, par un choc (détecté par la ceinture de sécurité). Le module avait attribué le numéro 0 à l'activité (information retournée par la réplique intermédiaire). La réplique finale a été attendue en mode bloquant.*

G^{en}$_{o}$M ne procède pas à des vérifications préalables quant à la viabilité du module, qui dépend essentiellement des capacités en calcul et en mémoire de la CPU. Par contre il offre différents moyens pour contrôler le bon fonctionnement du module in-situ.

L'objet de ces tests est de valider les algorithmes des codels et de vérifier que leurs temps de traitement sont compatibles avec la dynamique du système contrôlé. En effet, *la réactivité d'un module ne dépend que de la durée d'exécution des codels.*

Ces mesures vont permettre de déterminer les dispositions à prendre en cas d'exécution trop lente:

- Modification des codels
  - en optimisant les algorithmes,
  - en redécomposant les codels en des codels plus petits pour réduire les temps de réaction (activités de type `serveur`).
- Utilisation d'un processeur plus puissant.
- Répartition des modules sur plusieurs processeurs.

Les tests vont concerner trois aspects: la disponibilité de la ressource mémoire, les caractéristiques temps réel (liées à la charge de la CPU) et la qualité des algorithmes:

Disponibilité de la ressource mémoire: Pour des raisons d'efficacité et de sûreté de fonctionnement, les modules générés par G^{en}$_{o}$M et les bibliothèques de communication ne procèdent à aucune allocation dynamique de mémoire: en dehors d'allocations qui se feraient au niveau des codels (à la charge du développeur), l'initialisation correcte du module suffit à se garantir de tout problème lié à cette ressource durant l'exécution.

La mémoire utilisée par chaque tâche est aisément mesurable par des fonctions offertes par les systèmes d'exploitation temps réel. Cette information permet de "tailler" au mieux cette ressource (champ `stack_size`).

Moyens d'analyse des caractéristiques temps réel et de la charge de la ressource CPU: Les systèmes d'exploitation proposent des moyens de mesure de la charge globale de la CPU et de la répartition de cette charge, tâche par tâche, qui donnent un premier aperçu de la puissance du processeur relativement aux codels à exécuter.

De façon plus spécifique, les temps d'exécution instantanés et maximums des tâches d'exécution sont exportés dans les posters de contrôle et peuvent être surveillés par des tâches externes.

Afin de procéder à une analyse plus fine de la répartition de la charge et des séquencements des activités, le contexte de développement de G^{en}oM offre le moyen de dater (à quelques microsecondes près) et d'enregistrer chacun des événements internes au module: les changements d'état des activités, les débuts et fin d'exécution des codels, les réceptions de requêtes et les émissions de répliques. On peut ainsi établir les chronogrammes précis des activités. Les chronogrammes des figures 1.11 page 93 et 1.12 page 93 de la partie III, qui résultent de ces mesures, montrent en particulier:

- **Le temps d'exécution des codels** qui est visualisé par la largeur des créneaux des phases d'exécution (réduits à des impulsions à l'échelle du premier chronogramme). On constate, pour le cas présenté, que cette consommation en temps est largement compatible avec les capacités du processeur[9].
- **Le temps de réaction** est le temps écoulé entre la réception de la requête (début de phase **INIT**) et le début de l'exécution, et entre l'instant de réception d'une interruption et la fin du traitement de la seconde activité.
- **Les décalages temporels entre les activités périodiques** (génération de la consigne, lecture de cette consigne et asservissement) obtenus par la directive `delay`. Le chronogramme montre que les temps de traitement sont suffisamment courts pour que les traitements s'effectuent effectivement séquentiellement.
- **Les effets de priorités relatives** qui sont visualisés sur le deuxième chronogramme: la phase de démarrage (**start**) de l'activité de génération de consigne a une durée supérieure à la période des activités mais cela ne perturbe pas l'exécution périodique de l'activité d'asservissement de priorité plus forte (directive `priority`).

Soulignons que les résultats de ces mesures sont strictement équivalents en simulation ou en exécution réelle. En effet, la simulation étant définie au niveau du robot logique, les modules et donc le séquencement des activités sont inchangés (à condition bien sûr d'utiliser le même type de processeur).

Moyens d'analyse des algorithmes La qualité des algorithmes ne sera validée que lorsqu'ils auront été confrontés à des données réelles (issues des capteurs le cas échéant). Cependant des mesures en simulation ou en réel permettront l'analyse et la mise au point de ces algorithmes.

Pour cela on peut utiliser des outils génériques tels que `StethoScope` qui permet de suivre l'évolution de variables quelconques (par exemple des données de la SDI/f), ou encore des outils plus spécifiques que nous avons développés tels que les interfaces graphiques $\mathcal{G}$ $r\mathcal{H}2$ ou *oper* qui visualisent l'évolution de variables exportées dans les posters d'exécution. Ainsi les profils de vitesse et la comparaison de la trajectoire de consignes et de la trajectoire

9. Il s'agit bien du temps d'exécution et non pas du temps CPU: durant son exécution, cette tâche aura à partager le temps CPU avec des tâches plus prioritaires.

exécutée des figures III. 2.4 page 107 et III. 2.2 page 103 ont été obtenus par StethoScope. L'interface graphique de contrôle $\mathcal{GrH}_2$ visualisé sur la figure III. 1.19 page 99 a produit les figures concernant le recalage III. 1.17 page 96, les trajectoires planifiées III. 2.5 page 108 et III. 2.6 page 109, les trajectoires visualisées par caméras et selon les données odométriques du robot III. 2.10 page 114 et E.2 page 171, le recalage III. 1.17 page 96. Les échos ultrasons et l'évitement d'obstacles des figures III. 1.8 page 90 et III. 1.15 page 95 ont été obtenus par *oper*. Ces outils peuvent être utilisés indifféremment en simulation ou durant une expérimentation.

3.4 Conclusion

G^{en}oM permet de décrire un module, au moyen d'un langage de spécification adapté aux applications robotiques (en termes de requêtes, de posters et d'activités); de générer automatiquement ce module et ses fonctions d'interactions; de diriger son intégration et procéder à des tests préliminaires d'évaluation des codels.

Le seul aspect relatif à l'implémentation qui reste à la charge du développeur est la répartition des activités en tâches d'exécution. Nous envisageons d'automatiser cette procédure. G^{en}oM gérerait alors entièrement la répartition et le séquencement des activités. De ce point de vue, G^{en}oM s'apparente à un système d'exploitation qui séquencerait non pas des tâches mais des activités qui interagissent par requêtes et par posters. On pourrait même envisager (mais ce n'est pas l'objet de notre étude) de se dispenser complètement de système d'exploitation du commerce.

Les modules produits sont des systèmes réactifs qui, grâce à leur observabilité et à leur programmabilité, composent sous l'impulsion du superviseur d'exécution, un système délibératif: les services offerts par la couche fonctionnelle forment, du point de vue du superviseur, l'ensemble des actions élémentaires que peut accomplir robot, et dont l'exécution coordonnée et contrôlée permet de tendre vers le but global à atteindre.

Les règles qui régissent cet exécutif sont l'expression:

- des conflits déclarés au sein des modules (`incompatible_with`),
- des ressources partagées entre les activités des différents modules (`resources`),
- des relations client/serveur (`cs_client_from`) [10],
- et du type des activités (`activity_type`).

Actuellement cette information n'est pas exploitée explicitement (le programmeur les prend en compte implicitement avec les risques d'oubli ou d'erreur que cela sous-entend). Nous envisageons de procéder à une analyse globale des descriptions formelles des modules utilisés pour une application donnée afin d'établir ces règles automatiquement. On tendrait alors vers une génération automatique d'un exécutif dont le noyau peut être obtenu par compilation de ces règles (ce qui conduit à la génération des automates des modules), par exemple au moyen de Kheops.

10. Il sera peut être nécessaire de préciser le ou les services requis auprès du serveur par l'activité.

Troisième partie

Integrations et expérimentations

Chapitre 1

Présentation de la couche fonctionnelle d'Hilare2

La difficulté, mais aussi l'intérêt, de concevoir une architecture de contrôle pour un robot mobile, réside dans le non-déterminisme des interactions avec l'environnement d'évolution qui sont de nature faiblement prédictibles et d'occurrence aléatoire. Cette constatation rend indispensable une validation par de nombreuses expérimentations sur des machines réels.[1]

Le formalisme présenté dans les chapitres précédents et les techniques d'intégration de modules ont été très largement utilisés sur plusieurs robots. Nous allons décrire la couche fonctionnelle du robot *Hilare2*, un des robots expérimentaux de la famille *Hilare*. La présentation de cette couche fonctionnelle est particulièrement illustrative car elle intègre de nombreuses fonctionnalités.

Nous allons décrire dans un premier temps la plateforme mécanique et les équipements embarqués, puis le robot logique qui permet d'interfacer le robot physique et la couche fonctionnelle en la rendant indépendante de la plateforme matérielle.

Nous présenterons alors les modules de base de la couche fonctionnelle qui interviendront dans la majorité des applications. Une description détaillée du premier d'entre eux permettra d'illustrer le langage de spécification associé à G^{en}oM par un exemple concret. L'étude d'un second module, client d'un serveur, permettra d'analyser les séquencements d'activités mère/fille. Le fonctionnement temps réel de la couche fonctionnelle sera étudié à l'aide de chronogrammes d'exécution des activités.

La couche fonctionnelle ainsi composée offre des services de base mais aussi des services élaborés qui autorisent dès lors une commande sûre et conviviale du robot par un opérateur grâce à une interface graphique.

Des exemples d'intégration de la couche fonctionnelle dans une architecture de contrôle embarquée qui confère au robot son autonomie seront présentés aux chapitres 3 et 4: les expériences EDEN (navigation en environnement naturel) et STRADA (interaction de plusieurs robots mobiles).

1. Précisons tout de même que la simulation, plus facile à mettre en œuvre et dont la répétabilité aide à la mise au point, reste une étape utile, voire nécessaire.

1.1 Description du robot Hilare2

Les robots de la famille *Hilare* ont été conçus dans le but d'étudier et de développer le concept d'autonomie décisionnelle et opérationnelle d'une machine mobile. Cela implique des capacités de perception, de modélisation, d'identification, de décision et de coordination d'exécution.

Les capacités opératoires du robot dépendent d'abord de ses capacités à appréhender son environnement et donc du nombre et de la nature des capteurs dont il dispose. Afin d'expérimenter un large éventail de fonctions dans différents contextes expérimentaux, de nombreux capteurs sont intégrés, ou potentiellement intégrables, sur ces robots.

Quatre robots ont été développés dans cette optique: *Hilare1*, *Junior*, *Hilare2* et *H2bis*. Cependant, seuls *Hilare2* et *H2bis*, de conception plus récente, offrent toute la souplesse nécessaire pour intégrer de nouvelles fonctionnalités. Selon les travaux en cours, telles ou telles combinaisons de capteurs sont installées sur ces robots. Le plus complet actuellement est *Hilare2*, que nous allons présenter maintenant (quand à *Hilare1*, l'ancêtre, il est sur cales - avis aux musées ès-technologie -).

Etant donné que leur système de locomotion ne leur permet d'évoluer qu'en terrain plat, ces robots sont essentiellement des robots d'intérieur. Nous avons également mené des travaux en extérieur, sur terrain accidenté avec le robot *Adam* qui sera présenté au chapitre 3.

1.1.1 Le robot physique

Le robot *Hilare2*, tout comme ses 3 collègues, est un véhicule non-holonome mû par deux roues motrices indépendantes. La motorisation est assurée par des moteurs électriques à courant continu dont la rotation est transmise à des réducteurs mécaniques sans jeu. Ces moteurs sont commandés en tension par des variateurs qui effectuent une régulation analogique sur le retour d'une génératrice tachymétrique. Quelques caractéristiques:

- Dimensions: L 1.30 m, l 0.70 m, h 0.80 m, poids 400 kg (dont 200 kg de batteries)
- Vitesse maximale: 1 m/s
- Autonomie: 4h à 5h (batteries de 48V, 150 Ah)

Les capteurs et l'électronique de commande ont été installés, et pour certains conçus, au LAAS [Bauzil 92]. On peut citer pour l'essentiel:

- Des codeurs optiques sur les roues motrices (29.500 pas par tour de roue).
- Des roues odométriques placées sur l'axe des roues motrices (118.000 pas par tour de roue).
- Un gyroscope directionnel pour la mesure du cap.
- Un télémètre laser 3D.
- 32 télémètres à ultrasons.
- Un arceau de sécurité de détection de collision sur toute la périphérie.
- Un motif de diodes infrarouges sur le plateau supérieur pour une localisation externe par caméras.
- 2 accéléromètres (utilisables en mode inclinomètre).
- 3 caméras vidéo.
- Des capteurs infrarouges d'accostage de part et d'autre du robot.
- Un ensemble caméra/nappe laser.

Les quatre derniers éléments ne sont pas utilisés dans les travaux présentés ici.[2]

1.1.2 Le robot logique

Le robot logique est un ensemble d'interfaces logicielles entre le robot physique et la couche fonctionnelle, qui permet d'abstraire ses modules des aspects matériels des capteurs et des actionneurs commandés. Ce découplage permet de faire évoluer le matériel sans modifier les modules et surtout de rendre les modules portables sur différents robots. Ainsi, les modules d'*Hilare2* et d'*H2bis* sont identiques. Les fonctions du robot logique sont décrites dans [Bauzil 92].

Ce principe permet également de procéder à des simulations à un très bas niveau: la commande des capteurs et des actionneurs, intégré aux fonctions du robot logique, peut être remplacée par des algorithmes de simulation de commande. Là encore, les modules pourront être utilisés tels quels et pourront être testés intensivement avant d'être embarqués sur le robot. Les fonctions de simulation seront présentées par la suite.

1.1.3 Architecture informatique

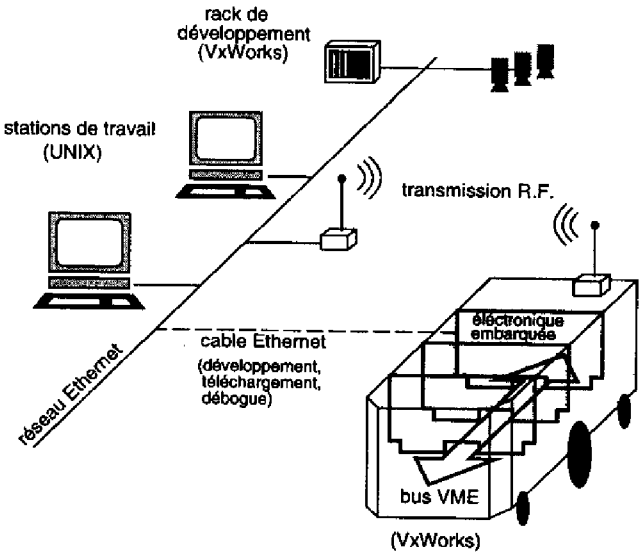


FIG. 1.1 – **Structure informatique générale.**

L'architecture informatique d'un robot expérimental doit allier des caractéristiques temps réel, un contexte d'intégration souple et des outils de développement puissants. Elle est organisée en deux domaines distincts, l'un constitué par un ensemble de stations de travail UNIX, l'autre par l'électronique embarquée du robot.

Le domaine UNIX, composé des stations de travail du laboratoire reliées par un réseau Ethernet, offre de nombreux utilitaires incluant des compilateurs, des éditeurs, des débogueurs, des facilités de communication et de nombreux outils conviviaux (graphiques...) qui vont permettre de développer et de tester les fonctionnalités qui seront embarquées.

2. Si des fonctions de vision ont été utilisées sur les robots *Junior* et *Hilare1*, elles n'ont pas eu d'application sur le robot *Hilare*. On présentera par contre une application sur le robot d'extérieur *Adam*.

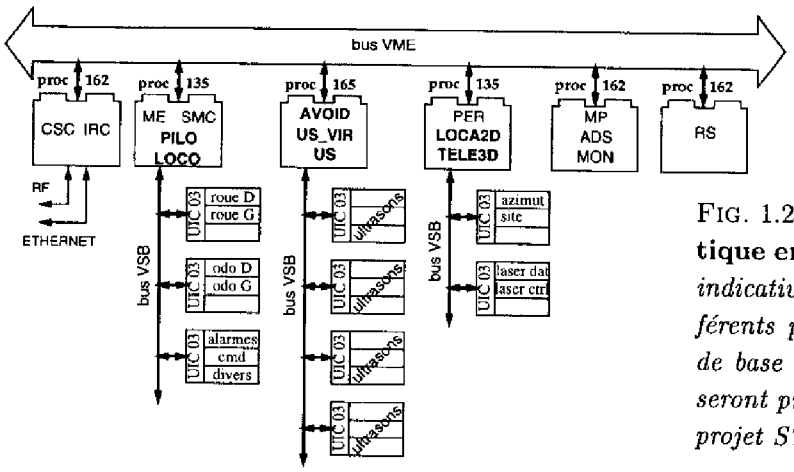


FIG. 1.2 – **Structure informatique embarquée** *et répartition indicative des modules sur les différents processeurs. Les modules de base sont en gras. Les autres seront présentés dans le cadre du projet STRADA.*

Sur le robot, nous souhaitions disposer d'un système d'exploitation qui fournisse des fonctions temps réel de gestion de tâches, de sémaphores, de communication multi-processeurs, *etc.*, avec des temps de réponse et des temps de commutation rapides. Le choix de l'équipe RIA du LAAS s'est porté sur le système d'exploitation VxWorks, développé par Wind River Systems. VxWorks a été conçu dans l'optique de profiter des outils de développement d'UNIX et de la connexion au réseau Ethernet pour bénéficier d'un environnement de développement et de supervision puissant à côté d'un système d'exploitation temps réel performant. Afin de disposer d'un système physique ouvert et reconfigurable, l'électronique embarquée est une architecture multi-processeurs à plusieurs bus.

Nous disposons de deux modes d'interaction entre les deux domaines: une connexion par câble "Ethernet" utilisée durant la phase de développement et de mise au point des logiciels embarqués, et une liaison hertzienne à 9600 bauds qui permet, sur la base du protocole SLIP, de communiquer avec le robot durant l'exécution des missions. Les figures 1.1 et 1.2 décrivent l'ensemble du système informatique.

1.2 La couche fonctionnelle d'Hilare2

Nous allons présenter les modules de base de la couche fonctionnelle d'*Hilare2*. Nous ne considérons ici que les aspects structurels et opératoires des modules, l'algorithmique étant abordée dans le chapitre 2.

Les huit modules qui font l'objet de cette présentation sont les suivants:

Module		Fonction
LOCO	Locomotion	Calcul de la position et asservissement du véhicule.
US et US_VIR	Ultrasons	Mesures proximétrique par ultrasons.
TELE3D	Télémètre laser	Acquisition d'images de points 3D.
PILO	Pilotage	Calcul et exécution de trajectoires.
AVOID	Evitement	Détection et évitement dynamique d'obstacles.
LOCA2D	Localisation	Modélisation 2D et recalage sur amers.
PLANIF2D	Planification	Planification de trajectoires.
LOCEXT	Loc. externe	Recalage absolu par caméras externes.

Les modules LOCO, US et TELE3D interagissent directement avec le robot logique et ne sont clients d'aucun autre module. Les modules PILO, AVOID et LOCA2D utilisent les services

des trois précédents. Le module PLANIF2D est uniquement un serveur. Le module LOCEXT a la particularité de ne pas être embarqué sur le robot.[3] Afin d'illustrer le développement et la description formelle d'un module, nous détaillerons plus particulièrement le module LOCO. Une mise en œuvre d'activités composées sera décrite à l'occasion de la présentation du module PILO, client du module LOCO.

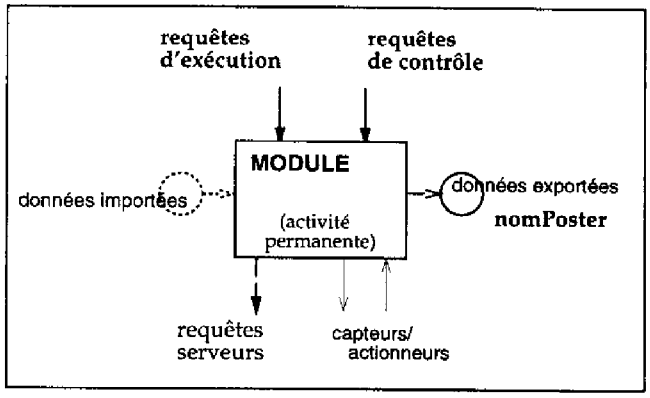


FIG. 1.3 – *Les interactions d'un module: notations.*

Les principales interactions de chaque module seront résumées par un schéma dont les notations sont exprimées sur la figure 1.3.

Ces différents modules de la couche fonctionnelle peuvent être visualisés sur la figure 1.4. Les transferts de données et les relations client/serveur seront décrits au fur et à mesure de la présentation des modules.

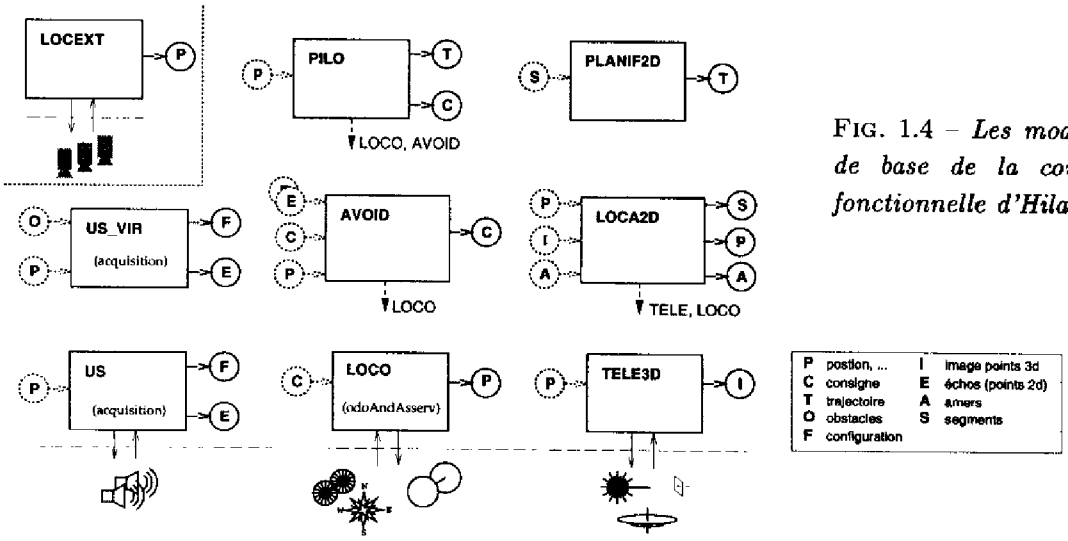


FIG. 1.4 – *Les modules de base de la couche fonctionnelle d'Hilare2.*

3. Ces modules ont été développés par différentes personnes: H. Bullata, R. Ferraz de Camargo, M. Herrb, M. Khatib et moi-même. Pour plus d'information concernant les algorithmes on trouvera des références concernant ces travaux au fur et à mesure de la présentation.

1.2.1 Le module locomotion LOCO

Le module LOCO a la charge de l'asservissement du véhicule, du calcul de sa position grâce à des capteurs proprioceptifs et d'autres services relatifs à la locomotion. Nous allons montrer comment s'intègrent ces fonctionnalités de base et comment on peut en modifier le fonctionnement selon les besoins de l'application. La présentation de ce module comporte deux parties: nous allons d'abord exposer ses fonctions principales, puis nous montrerons comment il a été intégré à l'aide de G^{en}oM. Les autres modules seront présentés plus rapidement.

1.2.1.1 Les services offerts par le module locomotion

▷ **Le calcul de la position:** Le robot dispose de capteurs proprioceptifs qui permettent de déterminer la position du véhicule à une fréquence suffisamment élevée pour pouvoir être utilisée dans la boucle d'asservissement.

On déduit des rotations des roues, mesurées par deux jeux indépendants d'odomètres (sur roues motrices et sur roues odométriques), les déplacements instantanés linéaires et angulaires du véhicule et, par intégration, sa position. L'inconvénient de cette technique est l'accumulation des erreurs au cours des déplacements, amplifiées par l'incertitude angulaire[4]. L'orientation du robot peut aussi directement être obtenue par un gyroscope dont l'incertitude n'est pas corrélée avec les déplacements, mais qui a par contre tendance à dériver au cours du temps.

Ces deux méthodes de calcul de la position sont donc complémentaires. L'usage de l'une ou l'autre méthode, ou de combinaisons de ces méthodes, peut être imposé par une requête de contrôle. Par défaut, on utilise le gyroscope lorsque le robot est en mouvement (*i.e.* vitesses supérieures à un seuil donné), et exclusivement l'odométrie le reste du temps[5]. Cependant il peut être intéressant d'imposer un choix, par exemple pour procéder à des calibrations ou à des études comparées, ou encore pour introduire volontairement des erreurs importantes de positionnement afin d'éprouver des procédures de recalage[6].

Cette localisation proprioceptive, ainsi que le calcul probabiliste de l'incertitude en position, sont exécutés par une activité périodique permanente. La position peut être recalée au moyen d'une requête de contrôle.

▷ **L'asservissement:** L'asservissement en position, qui produit une commande en vitesse et dont la loi sera présentée au chapitre suivant, est également assuré par une activité permanente. La consigne sur laquelle le véhicule s'asservit se trouve dans la SDI/f. Cette activité a également la charge de détecter les alarmes qui remontent du robot physique (niveau des batteries, choc arceau, arrêt d'urgence, surcouple, commande en mode manuel). On peut modifier par requête de contrôle certains paramètres de la loi d'asservissement.

▷ **Le suivi de consigne:** L'intérêt de l'asservissement est de suivre au mieux une trajectoire donnée, qui peut être un chemin planifié, un contournement dynamique d'obstacles ou le suivi d'un objet. La consigne, instationnaire, est déterminée par le module adéquat et exportée dans un poster. Une activité, démarrée sur requête d'exécution, permet de lire un

4. Intuitivement, l'erreur angulaire produit un cône d'incertitude sur la position qui va s'ouvrir de plus en plus au cours du déplacement.

5. La commutation s'opère dynamiquement par recalage mutuel.

6. L'odométrie "motrice" est de qualité relativement médiocre mais elle est moins coûteuse à installer.

poster spécifié en paramètre, de contrôler la validité de la consigne puis de la copier dans la SDI/f d'où elle sera lue par l'activité d'asservissement. Le transfert est effectué à la fréquence de l'asservissement avec une légère avance. Cette activité de type filtre ne peut se terminer que sur une interruption qui peut être soit demandée par un client, soit provoquée par un arrêt d'urgence détecté par l'activité d'asservissement (choc arceau, bouton d'arrêt d'urgence). L'activité doit alors s'assurer, durant la phase **inter** ou **end**, que la dernière consigne enregistrée dans la SDI/f va stabiliser le robot (vitesses nulles).

▷ **Les services satellites** Le module offre la possibilité de recaler la position et l'incertitude du robot, de modifier les gains d'asservissement, d'activer/désactiver des signaux sonores ou visuels, de modifier les paramètres géométriques ou cinématiques du véhicule, de procéder à un arrêt d'urgence, de connaître l'état des alarmes (niveau des batteries, choc arceau, arrêt d'urgence, mode de mobilité), ... Ces services sont disponibles via des requêtes de contrôle.

1.2.1.2 La déclaration du module locomotion

Afin d'illustrer la déclaration formelle d'un module, nous allons maintenant décrire celle du module LOCO dont la structure est présentée et les principales interactions sont illustrées par les figures 1.5 et 1.6.

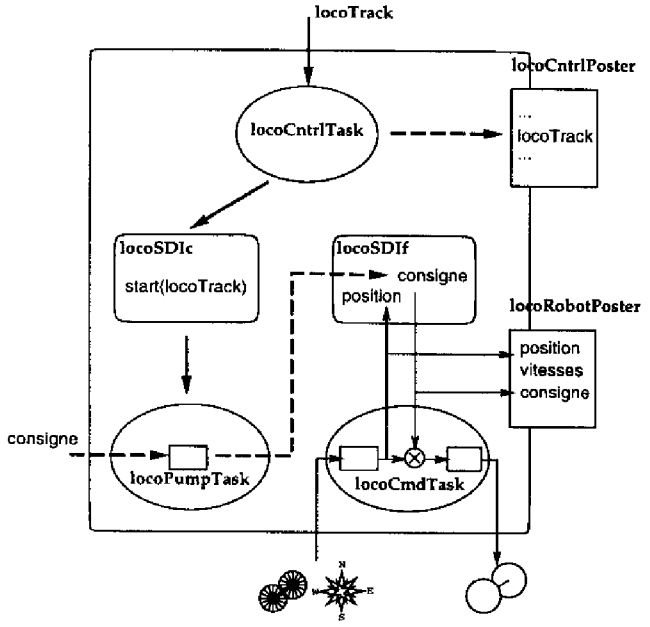


FIG. 1.5 – *Structure du module* LOCO *et exécution de l'activité* `locoTrack`. *Les flèches en gras correspondent au flux engendré par la requête. Le flux dû à l'activité permanente est en traits fins.*

▷ **Déclaration des activités permanentes** Les deux activités permanentes ont la même période et doivent s'exécuter séquentiellement (calcul position puis asservissement), il est donc naturel de leur associer la même tâche d'exécution. Dans la déclaration ci-dessous ces deux activités sont intégrées dans le codel nommé `odoAndAsserv`. Le codel `initOdoAndAsserv` invoqué au démarrage de la tâche initialise la SDI/f (position initiale, paramètres géométriques et cinématiques, *etc.*). Cette tâche d'exécution primordiale a la priorité la plus élevée du système. La période de 5 tics correspond à 25 ms.

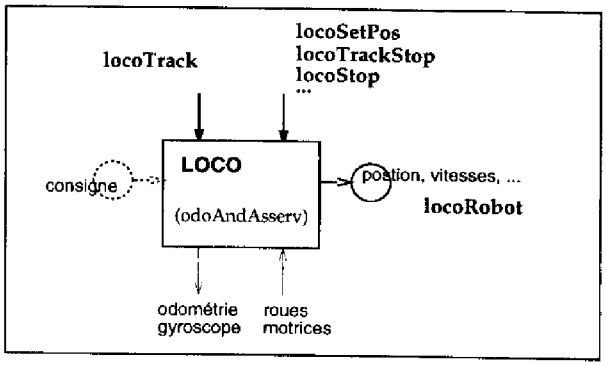


FIG. 1.6 – *Les principales interactions du module* LOCO.

```
exec_task CmdTask {
    period:        5;
    delay:         2;
    priority:      0;
    stack_size:    8000;
    c_init_func:   initOdoAndAsserv;
    c_func:        odoAndAsserv;
    resources:     wheels, odo, gyro, alarm;
};
```

▷ **Déclaration de la requête d'exécution** L'unique requête d'exécution proposée par le module LOCO est le suivi de consignes: `locoTrack`. Le nom du poster de consignes est transmis en argument de la requête (`input`). La phase d'initialisation d'exécution **start** vérifie l'existence du poster. La consigne est alors récupérée périodiquement (phase **exec**). Il ne peut y avoir qu'une instance de cette activité à un instant donné (`incompatible_with`), une seconde requête interromprait la première et poursuivrait le tracking sur un nouveau poster. Cette activité est un filtre à terminaison contrôlée: elle se termine soit à la demande d'un client (phase **INTER**), soit sur une défaillance interne (phase **end**). Dans l'un ou l'autre cas, le même codel `smoothStopTrack` stabilise le robot. Les différents possibilités de défaillance sont exprimées dans le champ `fail_msg`.

```
request Track {
    type:                exec;
    activity:            filter;
    input:               posterName::trackedPoster;
    c_exec_func_start:   findTrackedPoster;
    c_exec_func:         pumpReference;
    c_exec_func_end:     smoothStopTrack;
    c_exec_func_inter:   smoothStopTrack;
    exec_task:           PumpTask;
    fail_msg:            POSTER_NOT_FOUND, INVALID_REFERENCE,
                         BELT'SHOCK, EMERGENCY_STOP,
                         MANUAL_MODE, LOW_LEVEL_BATTERY;
    incompatible_with:   Track;
};
```

▷ **Déclaration des requêtes de contrôle:**

- **Accès aux données de la SDI/f:**
  - `locoGetPosType` - choix des capteurs proprioceptifs,
  - `locoGetAlarm` - état des alarmes,
  - `locoGetGeoConfig` - paramètres géométriques,
  - `locoGetCmdConfig` - paramètres d'asservissement (gains du PI et bornes sur les vitesses et accélérations).

  La déclaration de ces requêtes ne comporte que deux champs: le type de la requête et la donnée attendue, par exemple:

```
request GetCmdConfig {
    type:    control;
    output:  commandParameters::cmd;
};
```

- **Modification de données de la SDI/f:**
  - `locoSetPosType` - choix des capteurs proprioceptifs.
  - `locoSetCmdConfig` - paramètres d'asservissement,
  - `locoSetGeoConfig` - paramètres géométriques,
  - `locoSetPos` - recalage de la position et des incertitudes,
  - `locoGoTo` - changement de la consigne (a pour effet d'asservir le véhicule sur une nouvelle consigne et donc de le faire bouger selon la loi d'asservissement),
  - `locoBipbip` - activation des signaux visuels et sonores.

  Le paramètre à enregistrer est indiqué dans le champ `input`. Une fonction de contrôle peut être définie pour vérifier la pertinence de ce changement. Dans l'exemple ci-dessous le codel `controlCmd` vérifie la validité des paramètres et retourne le bilan `S_locoCntrlTask_INVALID_PARAMETERS` en cas de problème:

```
request SetCmdConfig {
    type:            control;
    input:           commandParameters::cmd;
    c_control_func:  controlCmd;
    fail_msg:        INVALID_PARAMETERS;
};
```

- **Interruption de l'activité `locoTrack`:**
  - `locoTrackEnd` - arrêt explicite de l'activité de suivi de consigne,
  - `locoStop` - arrêt d'urgence.

  Ces deux requêtes, qui ont pour effet d'interrompre l'activité `locoTrack`, sont déclarées incompatibles avec celle-ci. La requête `locoStop` dispose de plus d'une fonction de contrôle qui, en commutant la valeur d'un booléen de la SDI/f, transmet l'ordre d'arrêt d'urgence à l'activité d'asservissement, ce qui permet de s'arrêter quelle que soit la cause du mouvement (requête `locoGoTo` par exemple).
  La requête `locoTrackEnd` est strictement équivalente à la requête standard `locoAbort` mais ne requiert pas l'identificateur de l'activité.

```
request TrackEnd {
    type:                control;
    incompatible_with:   Track;
};
request Stop {
    type:                control;
    c_control_func:      Stop;
    incompatible_with:   Track;
};
```

▷ **Déclaration des tâches d'exécution** Le module LOCO dispose de deux tâches d'exécution. La tâche `locoCmdTask` de priorité importante ne s'occupe que de l'activité permanente `odoAndAsserv` (voir sa déclaration dans le paragraphe traitant des activités permanentes). L'activité de suivi de consigne est prise en charge par une seconde tâche: `locoPumpTask`.

▷ **Déclaration des posters** La position, son incertitude probabiliste et la consigne courante sont exportées à la fréquence de l'asservissement dans le poster `locoRobot`:

```
poster Robot {
    update:     auto;
    data:       Position::posCntrl.robot,
                PosError::posCntrl.odoError,
                Ref::ref;
    activity:   odoAndAsserv::exec;
};
```

▷ **Déclaration de la SDI/f:** Elle dispose de champs pour enregistrer les paramètres des requêtes et des répliques ainsi que pour les données exportées dans le poster `locoRobot`.

```
#include "locoStruct.h"        /* Définition des structures */
#include "locoConst.h"         /* Valeurs par défaut de la SDI/f (vitesses max., PI, ...) */
typedef struct LOCO'STR {
 POS_CNTRL_STR posCntrl;       /* Position asservie */
 REF_STR ref;                  /* Consigne courante */
 GEO_PARAM_STR geo;            /* Paramètres géométriques */
 CMD_PARAM_STR cmd;            /* Paramètres d'asservissement */
 SET_POS_STR setPosStr;        /* Structure de recalage */
 ...
} LOCO_STR;
```

1.2.2 Les modules ultrasons US et US-VIR

Les obstacles dans l'environnement du robot sont détectés de façon dynamique à l'aide de capteurs ultrasons disposés à la périphérie du robot, qui mesurent la distance qui les sépare de l'obstacle le plus proche. Le module US gère cette ceinture d'ultrasons.

Toutes les 50ms, l'activité permanente `acquisition` excite les ultrasons, attend le retour des échos, en déduit la distance libre, puis exporte cette donnée dans le poster `usEchos` sous forme de coordonnées cartésiennes dans les deux repères liés au robot et à l'environnement (pour le calcul dans le repère global, la position du robot est lue dans le poster `locoRobot`).

Le choix des capteurs ultrasons excités et le séquencement de ces excitations (excitations simultanées: signal puissant mais couplage des échos, excitations séquentielles: pas de couplage mais la fréquence des mesures pour un capteur donné est plus faible) peuvent être redéfinis par la requête de contrôle `usSelect`. Cette "table d'excitation" enregistrée dans la SDI/f va guider l'activité permanente.

Des activités de surveillance seuils sur les données proximétriques peuvent être mises en place par la requête d'exécution `usMonitor`.

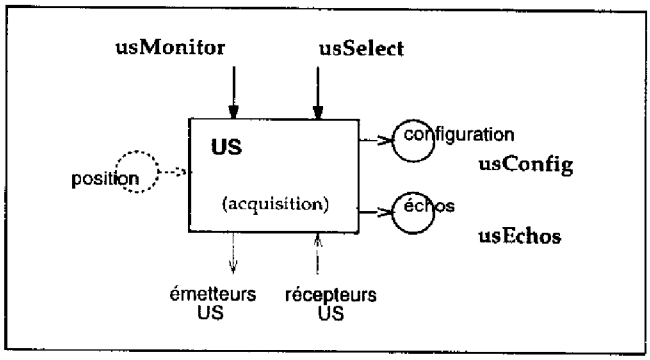


FIG. 1.7 – *Les principales interactions du module* US.

Un second module nommé US-VIR calcule des échos sur des obstacles virtuels et produit un poster `usVirEchos` dont la structure est identique à celle du poster `usEchos`. Les obstacles virtuels sont mis à jour par la requête d'exécution `usVirSetObstacles` qui lit leur description géométrique dans un poster spécifié en paramètre de la requête.

Ces échos, qui pourront être intégrés au même titre que les échos réels dans une procédure d'évitement d'obstacles (voir le module AVOID), ont différentes applications:

- ils permettent de contenir le robot dans une zone délimitée par des murs virtuels qui correspondent par exemple à un lieu topologique de l'environnement (le robot ne sera pas libre d'aller n'importe où durant un évitement d'obstacles),
- ils permettent de modéliser les obstacles fixes de l'environnement autorisant ainsi le robot à manœuvrer au plus près de ces obstacles répertoriés, en réduisant les marges de sécurité relatifs échos virtuels qu'ils induisent.

L'algorithme de calcul des échos sur les obstacles virtuels peut être repris (en ajoutant un bruitage) pour simuler les échos réels (voir la section 1.3). La figure 1.8 obtenue par l'interface graphique *oper* montre les échos réels et les échos virtuels sur un obstacle modélisé.

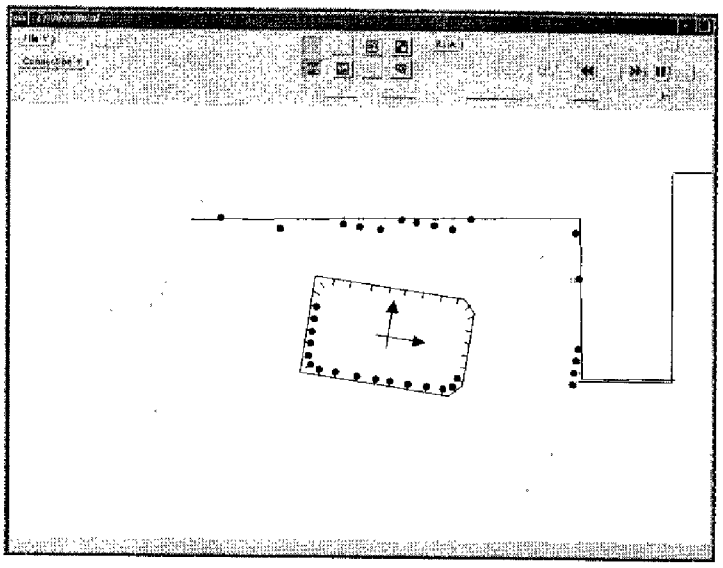


FIG. 1.8 – *Les échos réels (en grisé) et virtuels (en noir). L'absence d'écho pour un capteur donné est marqué par un point à l'emplacement de ce capteur. Les échos virtuels ont été volontairement bruités (ce qui en dehors de la simulation ne présente pas d'intérêt!).*

1.2.3 Le module télémètre laser TELE3D

Des données sur l'environnement du robot peuvent être acquises au moyen d'un télémètre laser et d'un dispositif de balayage à deux axes commandés par le module TELE3D qui permet d'acquérir des images de points 3D. Le rayon laser balaye l'environnement en site au moyen d'un miroir rotatif, l'ensemble du système étant disposé sur une plateforme dont la rotation engendre le balayage azimutal. Les acquisitions sont obtenues au moyen de deux requêtes d'exécution:

- `tele3dAcqui2d`: coupes horizontales 2D (dans le plan de son choix par orientation du miroir)
- `tele3dAcqui3d`: images de points 3D par balayage selon les deux axes.

Les points acquis sont enregistrés dans le poster `tele3dPoints`.

L'acquisition des points étant relativement lente (100Hz, soit 4s pour une coupe sur 360deg avec une discrétisation de 1deg), celle-ci s'opère à l'arrêt du robot. Un système plus rapide (jusqu'à 2000Hz) devrait être installé prochainement autorisant des acquisitions en mouvement. Les tirs laser, beaucoup plus directionnels que les échos ultrasons, seraient alors un complément efficace à la détection dynamique d'obstacles.

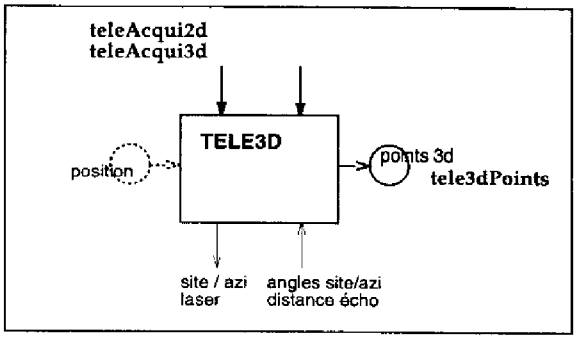


FIG. 1.9 – *Les principales interactions du module* TELE3D.

1.2.4 Le module pilotage PILO

En présence d'obstacles on ne peut rejoindre une configuration sous la simple impulsion de la loi d'asservissement (requête `locoGoTo`). Ce chemin devra être planifié en prenant en compte les obstacles connus et le robot devra rester sur la voie ainsi tracée ou dans des limites fixées. Afin que l'asservissement puisse effectivement suivre le chemin géométrique, la trajectoire dynamique devra respecter les contraintes cinématiques (équations non-holonomes) et les contraintes dynamiques (limitations en vitesses et en accélération) du véhicule. Ces trajectoires sont calculées et exécutées par le module PILO (les algorithmes sont présentés au chapitre suivant).

L'exécution consiste à produire et à exporter dans un poster une consigne instationnaire qui court le long de la trajectoire et sur laquelle devra s'asservir le véhicule. Le suivi de la consigne peut être demandé au module LOCO par la requête `locoTrack`, mais également au module AVOID présenté dans la section suivante par la requête `avoidTrack`. Le module PILO est donc client du module LOCO (ou du module AVOID) et l'activité d'exécution d'une trajectoire est mère de l'activité `locoTrack` (ou `avoidTrack`). Après la présentation générale du module PILO, nous montrerons à l'aide de chronogrammes comment se séquencent ces différentes activités.

▷ **Les requêtes du module pilotage** La requête de contrôle `piloAvoidOnOff` permet de sélectionner le mode normal (`locoTrack`) ou le mode "évitement" (`avoidTrack`).

La requête de contrôle `piloSlow` permet de modifier la vitesse de parcours d'une trajectoire pendant son exécution (pourcentage des vitesses maximales). On peut par exemple arrêter le véhicule en le maintenant sur sa trajectoire (vitesses nulles) puis repartir.

Le module PILO permet d'exécuter quatre types de trajectoire, dont le tracé initial et les écarts tolérés sont transmis en paramètre de la requête d'exécution correspondante: `piloMove` exécute des segments de droite, `piloTurn` exécute des arcs de cercles, `piloSmooth` exécute des lignes brisées, `piloReedsShepp` exécute des chemins de Reeds & Shepp[7]. Les deux derniers chemins n'étant pas exécutables tels quels à cause des discontinuités de tangence ou de courbure, l'activité devra procéder à une opération préalable de lissage[8].

La requête d'exécution `piloMonLength` permet d'installer des surveillances sur la passage

7. Les chemins de Reeds&Shepp sont composés de segments de droite et d'arcs de cercle qui sont produites par un planificateur de trajectoire pour véhicules non-holonomes à rayon de giration borné.

8. Les algorithmes de lissage, (voir la section 2.1.2), consistent à joindre des droites ou des cercles par des clothoïdes et/ou des anticlothoïdes dont les courbures varient continûment avec l'abscisse curviligne ou la variation angulaire.

d'abscisse curviligne sur la trajectoire.

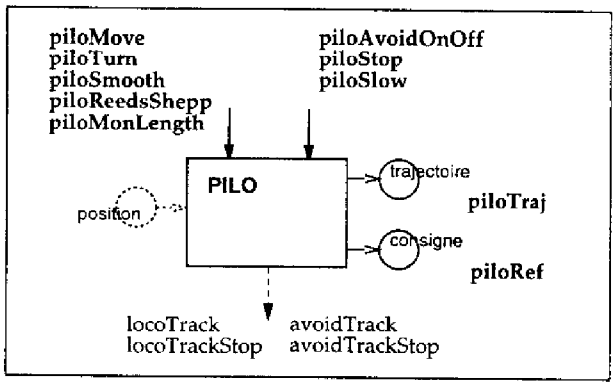


FIG. 1.10 – *Les principales interactions du module* PILO.

▷ **Le séquencement des activités** En dehors de la phase **start** qui utilise des algorithmes spécifiques à chaque type de trajectoire, les codels associés aux autres étapes sont communs aux quatre requêtes. Ces différentes phases et les relations établies avec le serveur (LOCO ou AVOID) sont les suivantes:

- La phase **start** calcule la trajectoire exécutable (lissage du chemin géométrique et calcul des profils de vitesses trapézoïdaux optimaux), puis démarre l'activité de suivi de consigne auprès du serveur concerné.
- La phase **exec** est périodique: le codel, commun aux quatre requêtes, parcourt la trajectoire calculée et détermine la consigne (en position et en vitesses) pour l'instant t+1. Cette consigne est mise à jour à chaque période dans le poster `piloRef` d'où elle peut être lue par l'activité fille.
- La phase **end**: l'activité d'exécution de trajectoire est un filtre auto-terminant qui se termine
  - soit avec la fin normale de la trajectoire. Elle interrompt alors l'activité fille qui est un filtre à terminaison contrôlée (requête `trackEnd`),
  - soit par une défaillance qui a causé l'interruption de l'activité fille qui envoie prématurément sa réplique finale. C'est ainsi que dans l'exemple de la figure II. 3.2 page 74 l'interruption de la requête `piloMove` fait suite à un choc de l'arceau de sécurité.
- La phase **INTER** est atteinte si l'activité est directement interrompue par un client, l'activité détermine alors une consigne d'arrêt puis interrompt l'activité fille.

Le chronogramme de la figure 1.11 page suivante a été obtenu expérimentalement grâce à l'enregistrement des événements internes (processeur 68040 à 25MHz). Il montre le déroulement normal de l'exécution d'une trajectoire: l'activité `piloMove` calcule la trajectoire, et démarre l'activité `locoTrack` (état **start**) avant d'exporter périodiquement la consigne dans le poster `piloRef` (état **exec**). A la même période mais avec un léger décalage, l'activité `locoTrack` enregistre cette consigne et la transmet, par la SDI/f, à l'activité permanente `odoAndAsserv` qui réalise alors l'asservissement. Lorsque la trajectoire est terminée, l'activité `piloMove` passe par l'état **end** où elle demande l'arrêt de l'activité `locoTrack` au moyen de la requête de contrôle `locoTrackEnd`. L'activité `locoTrack` transite alors temporairement par l'état **INTER**, puis se termine.

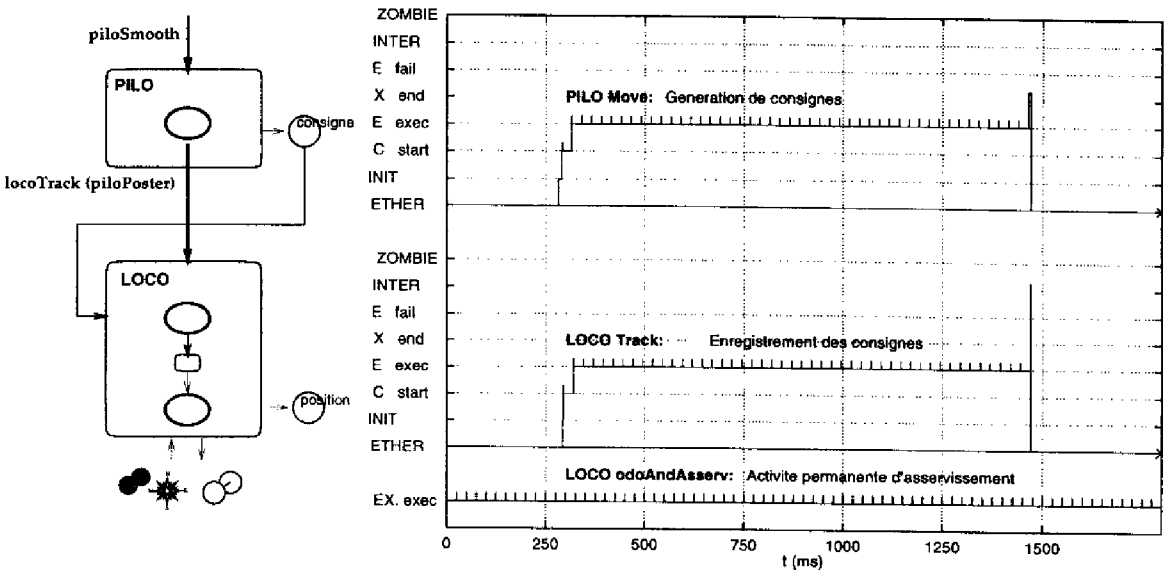


FIG. 1.11 – *Chronogramme des activités pendant l'exécution d'une trajectoire.*

Le second chronogramme est similaire au précédent mais à une échelle de temps différente qui permet d'apprécier les temps de traitement et les décalages entre les activités. La trajectoire calculée étant plus complexe (requête `piloSmooth` avec une ligne brisée composée d'une dizaine de points de rebroussement), l'activité est maintenue un certain temps dans l'état **start**. Par le jeu des priorités cela ne perturbe pas l'activité d'asservissement. Le temps de calcul et d'exportation de la consigne reste négligeable (état **exec**).

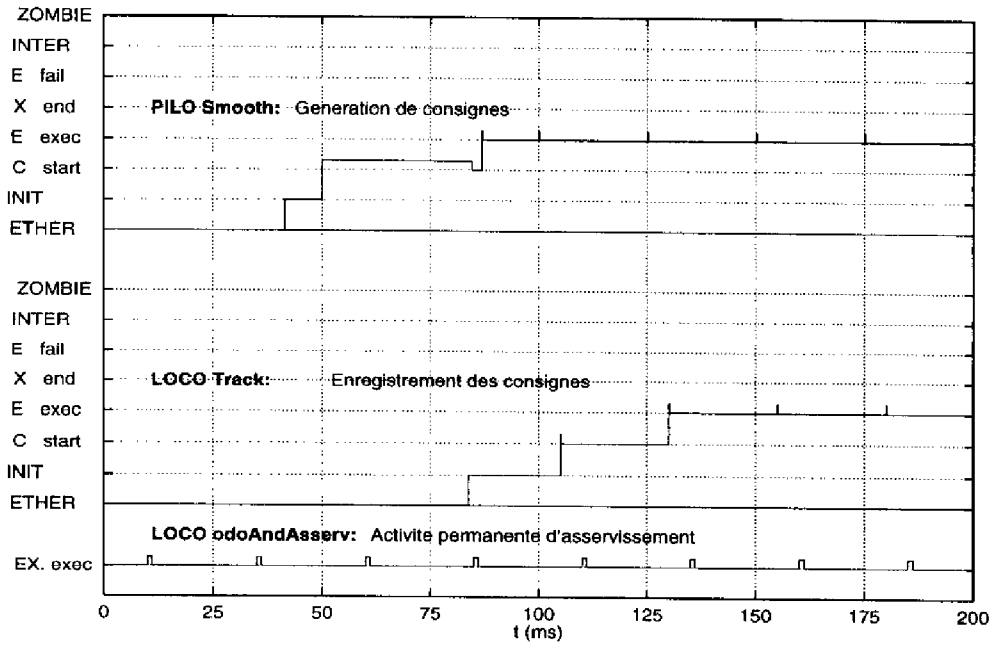


FIG. 1.12 – *Chronogramme des activités pendant l'exécution d'une trajectoire complexe.*

1.2.5 Le module évitement AVOID

Afin que le robot puisse évoluer en sécurité sans s'arrêter dès qu'un obstacle se présente, la couche fonctionnelle propose une fonction de déplacement avec évitement dynamiquement d'obstacles sur la base de données proximétriques. Ce service, assuré par le module AVOID, se fonde sur le principe des champs de potentiels [Khatib 86]: la consigne crée une force attractive et les obstacles des forces répulsives inversement proportionnelles à la distance au robot.

Les obstacles sont actuellement détectés au moyen des ultrasons, mais un télémètre laser assez rapide peut également remplir cet office. La méthode des potentiels a été étendue afin de faire dépendre la force répulsive de l'orientation de la normale à l'obstacle par rapport à la direction du mouvement du robot [Khatib 95]: par exemple si le robot avance parallèlement à un obstacle, celui-ci n'aura pas d'effet répulsif. Pour définir les normales aux obstacles, les points correspondants aux échos sont approximés localement par des segments.

▷ **Activités et requêtes** L'évitement consiste donc à suivre une consigne tout en évitant les obstacles grâce à des informations proximétriques fournies par exemple par le module US. La consigne peut être une position fixe transmise par la requête d'exécution `avoidGoTo`, ou un point instationnaire transmis via un poster spécifié par la requête d'exécution `avoidTrack`. On peut ainsi suivre une trajectoire calculée par le module PILO, ou encore une cible mobile localisée par un module adéquat. Une troisième requête d'exécution: `avoidFollowWalls` permet de suivre un mur modélisé à partir des données proximétriques. L'activité `avoidFollowWalls` calcule directement une consigne, exportée dans le poster `avoidRef`, et sur laquelle le robot va s'asservir au moyen de la requête `locoTrack`. Le séquencement des activités entre les modules AVOID et LOCO est identique à celui présenté entre les modules PILO et LOCO. La requête d'exécution `avoidMonLength` permet de surveiller le passage d'abscisse curviligne sur la trajectoire de consigne[9].

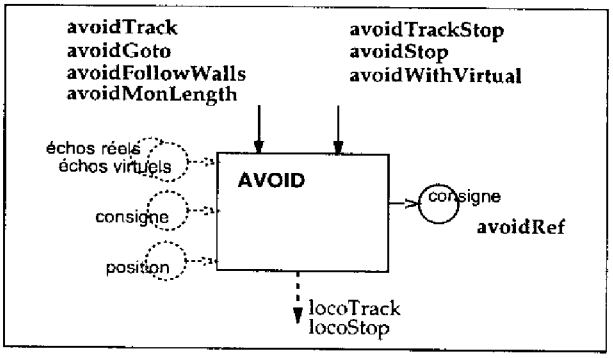


FIG. 1.13 – *Les principales interactions du module* AVOID.

La figure 1.14 montre la coopération entre les modules US, LOCO, AVOID et PILO lorsqu'une requête d'exécution de trajectoire à été requise auprès du module PILO en mode évitement: le module PILO produit des consignes qui sont filtrées par le module AVOID selon les données proximétriques du module US, puis transmises au module LOCO. Un résultat d'exécution est présenté sur la figure 1.15 page suivante.

9. La surveillance ne se déclenche que lorsque le robot est *sur* la trajectoire de consigne et au delà du point de passage surveillé ou lorsque l'exécution de la trajectoire est terminée, en ce cas un bilan spécifie la cause du déclenchement de la surveillance.

Afin d'empêcher le robot d'errer à la recherche de son chemin ou d'emprunter un mauvais chemin (par exemple passer à droite des obstacles de la figure 1.15 page ci-contre) on peut borner l'écart maximal à la trajectoire originale par la requête de contrôle `avoidMaxDev`, ou imposer des frontières via les murs virtuels du module US-VIR (`avoidWithVirtual`). Si un obstacle ne peut être contourné sans enfreindre ces limites, alors l'exécution s'interrompt avec le bilan `S_avoidCmdTask_IMPORTANT_DRIFT` ou `S_avoidCmdTask_BLOCKED` selon le cas. Généralement cela se traduira par une nouvelle modélisation de l'environnement suivie d'une nouvelle planification de chemin.

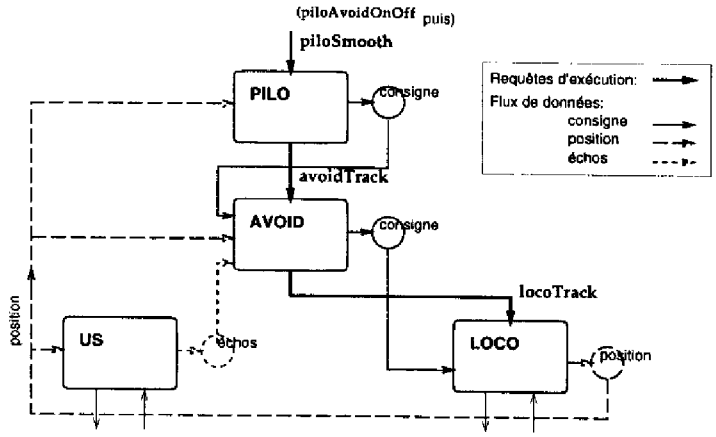


FIG. 1.14 – *Flux de contrôle et flux de données lors de l'exécution d'une trajectoire avec évitement local d'obstacles (mode avoid actif).*

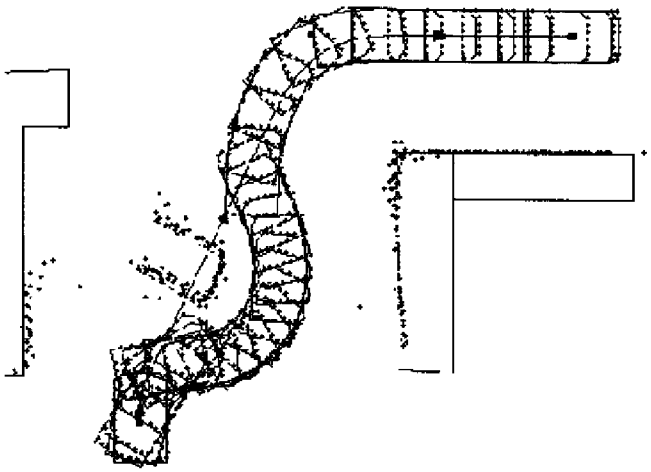


FIG. 1.15 – *Evitement d'un obstacle lors de l'exécution d'une trajectoire. La trajectoire de consigne est rejointe au plus tôt.*

1.2.6 Le module localisation extéroceptive LOCA2D

Le robot doit de temps à autre recaler sa position par rapport à son environnement et, quand cela est nécessaire, construire des modèles des obstacles avoisinants. Ces fonctions sont assurées par le module LOCA2D qui, à partir d'images de points laser acquises via le serveur TELE3D, extrait des segments. Ces segments permettent de modéliser les obstacles qui peuvent être ajoutés au modèle de l'environnement; des appariements entre ces segments perçus et les segments du modèle permettent de localiser le robot à l'aide d'un filtre de Kalman. Le filtre de Kalman généralisé ([Moutarlier 91]) réestime également les positions des

objets qui composent l'environnement et les incertitudes sur ces positions. On trouvera les détails techniques dans [Bullata 96].

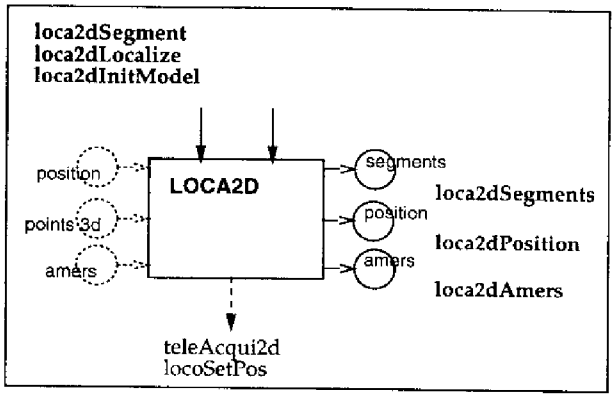
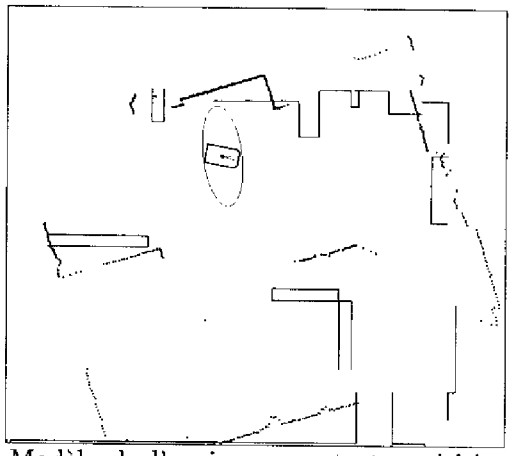


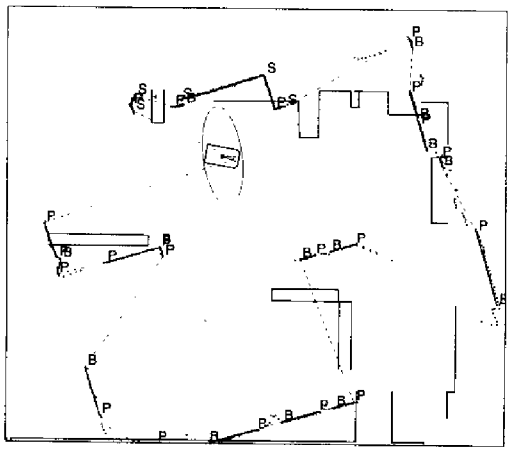
FIG. 1.16 – *Les principales interactions du module* LOCA2D.

La figure 1.17 présente les étapes successives d'acquisition, de segmentation et de recalage.

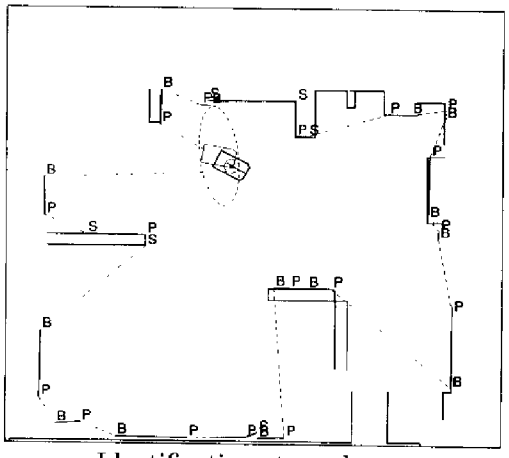


Modèle de l'environnement et aquisition laser.

FIG. 1.17 – **Localisation extéroceptive:** *La configuration estimée du robot et les incertitudes en position et en orientation sont visualisées sur la première image. Pour recaler le robot on procède d'abord à une acquisition de points par coupe laser (*`tele3dAcqui2d`*). Des segments sont alors extraits de cette image de points. L'appariement de ces segments avec le modèle préexistant de l'environnement permet de recaler le robot et de réduire l'incertitude sur cette position. Le modèle sera lui-même affiné. Cette séquence exécutée sur le robot a été visualisée au moyen de l'interface graphique GrH2.*



Extraction des segments.



Identification et recalage.

Trois requêtes d'exécution sont disponibles:

Trois requêtes d'exécution sont disponibles:

- `loca2dInitModel` permet d'initialiser un modèle de l'environnement,
- `loca2dSegment` acquiert une image de points laser grâce au module TELE3D et extrait les segments qui sont exportés dans le poster `loca2dSegments`,
- `loca2dLocalize` procède au recalage à partir des segments perçus. Les positions du robot et des objets ré-estimés sont exportés dans les posters `loca2dPosition` et `loca2dAmers`.

1.2.7 Le module de planification de trajectoire PLANIF2D

Ce module permet de trouver une trajectoire dans un environnement encombré. Les obstacles sont extraits du modèle de l'environnement ou déduits de la segmentation produite par la requête `loca2dSegments`. L'algorithme calcule tout d'abord, à partir d'une représentation "bitmap", le voronoï des obstacles locaux (*i.e.* l'ensemble des points équidistants à deux obstacles) que devra emprunter en partie le robot modélisé par son cercle circonscrit augmenté de l'incertitude en position. Puis est définie la trajectoire en joignant les positions initiale et finale au voronoï et en sélectionnant les arcs du diagramme de voronoï qui seront effectivement parcourus par le robot.[10]. La trajectoire est approximée par une ligne brisée et précise les écarts maximums tolérés qui seront considérés lors du lissage préalable à l'exécution par le module PILO.

La requête d'exécution `planif2dVoronoï` permet de précalculer le voronoï pour un ensemble d'obstacles décrits dans un poster dont les coordonnées sont transmises en paramètre de la requête. La requête d'exécution `planif2dTraj` calcule la ligne brisée et les tolérances associées, puis l'exporte dans le poster `planif2dTraj`.

1.2.8 Le module de localisation externe LOCEXT

Comme nous l'avons vu, la position du robot peut être déterminée par des capteurs embarqués proprioceptifs (intégration du mouvement: module LOCO) ou extéroceptifs (relativement aux objets de l'environnement: module LOCA2D). Une troisième possibilité qui permet de localiser le robot de façon absolue consiste à utiliser des capteurs externes au robot, liés à l'environnement.

Ceci permet de disposer d'une référence absolue permettant d'initialiser, de calibrer ou de valider les procédures de localisation embarquées. La localisation externe est assurée par le module LOCEXT qui, contrairement aux précédents, n'est pas embarqué sur le robot. Nous avons cependant choisi de définir ce système sous la forme d'un module afin de pouvoir accéder à ses services et aux résultats de ses traitements de la même façon que pour les éléments de la couche fonctionnelle.

Des diodes infrarouges disposées sur le plateau supérieur du robot composent un motif qui est identifié et localisé au moyen de trois caméras qui couvrent le champ d'évolution du robot. La technique de localisation est présentée au chapitre suivant, ainsi qu'une application à la calibration automatique de l'odométrie.

Les requêtes disponibles sont la localisation du robot `locextFind` (une seule itération) et le suivi du robot `locextTrack`. La position est exportée dans le poster `locextPosition`.

10. On présentera, dans le cadre du projet STRADA, un planificateur dans l'espace des configurations (x, y, θ) et pour lequel le robot est modélisé par un rectangle (4.2.5 page 137)

Les motifs de diodes, qui doivent être propres à chaque robot afin de les discriminer, sont enregistrés par la requête de contrôle `locextSetModel`.

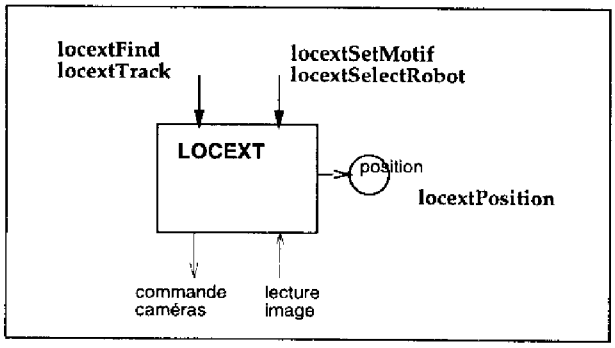


FIG. 1.18 – *Les principales interactions du module* LOCEXT.

1.2.9 Autres modules

Nous avons présenté les modules de base de la couche fonctionnelle d'*Hilare2* (figure 1.4 page 83) qui interviennent dans la majorité des expérimentations. D'autres modules ont été développés ou sont en cours de développement. On peut citer un module d'acquisition d'image vidéo (utilisé sur les robots *Junior* et *Adam*), un module pour effectuer des manœuvres de parking très contraintes qui nécessitent un asservissement relatif aux obstacles segmentés par LOCA2D, des planificateurs de chemins, *etc.*. On abordera d'autres modules dans les chapitres 3 et 4 qui ont été développés dans le cadre des projets EDEN et STRADA. En particulier, le projet EDEN fait intervenir des acquisitions d'images vidéo, des modélisations de terrains accidentés, des planifications de chemin "3D", et le projet STRADA utilise un module de planification multi-robots.

1.3 Simulation

La notion de robot logique nous a permis de développer une simulation complète du robot à très bas niveau qui permet d'utiliser les modules indifféremment sur le robot ou en simulation. Il suffit pour cela de substituer aux fonctions de commande des capteurs et des actionneurs du robot logique des fonctions d'émulation de ces commandes.

Ainsi, la fonction de commande des roues est remplacée par des fonctions qui émulent les rotations des roues et du mouvement relatif du gyroscope, les fonctions de lecture des codeurs odométriques et gyroscopiques retournant simplement les variations angulaires obtenues par les fonctions précédentes.

Les échos des ultrasons sont l'intersection entre le tir fictif d'un émetteur ultrason et un modèle "bitmap" ou polygonal d'obstacles. Le cône d'ouverture de ces capteurs est modélisé ($\simeq 30deg$) et un bruit blanc est ajouté pour disposer de données proximétriques réalistes. La technique d'émulation les tirs laser est identique si ce n'est que l'on suppose que le point d'impact est ponctuel (pas de cône d'ouverture).

Du point de vue de la couche fonctionnelle, rien ne distingue le robot vrai du robot émulé. L'intérêt de ces moyens de simulation est de pouvoir procéder à de nombreux tests préalables concernant les modules, leurs interactions, la dynamique des activités, la disponibilité en ressource mémoire et en temps CPU, et la validité et la fiabilité des algorithmes.

1.4 Interface utilisateur

Afin qu'un opérateur puisse commander directement le robot, une interface graphique qui s'adresse aux modules par le biais des bibliothèques d'interaction générées par G^{en}oM, a été développée. Ce graphique, nommé $\mathcal{GrH}_2$ permet:

- de visualiser la position du robot et son incertitude (poster `locoRobot`),
- de visualiser la position du robot selon les caméras (poster `locextPosition`),
- de visualiser les échos des ultrasons (posters `usEchos` et `usVirEchos`),
- de définir des obstacles virtuels et de visualiser les échos (poster `usVirEchos`),
- de saisir des trajectoires à la souris, de les prévisualiser[11], et de les exécuter en mode normal ou en mode évitement,
- de demander le calcul d'une trajectoire auprès de PLANIF2D, de la visualiser et de l'exécuter dans le mode de son choix,
- de modifier dynamiquement les vitesses d'exécution ou d'arrêter le robot,
- de requérir un recalage auprès de LOCA2D et de visualiser les segments extraits,
- de recaler le robot selon la position déterminée par les caméras (LOCEXT),
- ...

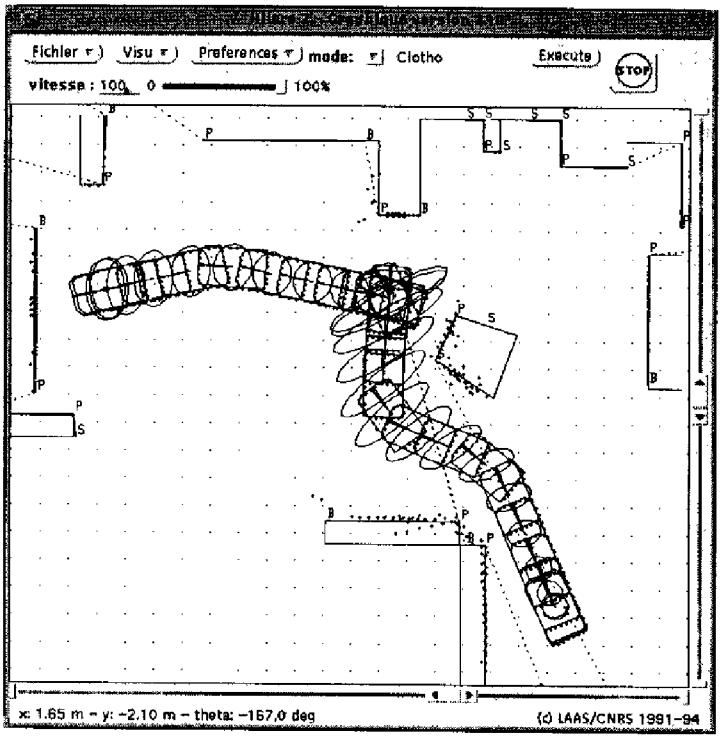


FIG. 1.19 – **L'interface graphique de contrôle $\mathcal{GrH}_2$.** *Le chemin initial a été saisi à la souris. Les échos ultrasons et l'évolution de l'incertitude sur la position sont visualisés. Avant de d'exécuter la rotation et une nouvelle trajectoire, l'opérateur a requis un recalage.*
Les opérations disponibles depuis cette fenêtre sont la saisie de chemins, la modification dynamique des vitesses, l'arrêt d'urgence, l'activation du mode évitement, l'adjonction d'obstacles virtuels, ... et l'ouverture des fenêtres de contrôle annexes.

Cette interface ne s'adressant qu'à la couche fonctionnelle s'utilise indifféremment en simulation ou avec les robots. Au delà d'un outil de mise au point puissant, ce système montre que la couche fonctionnelle est, au moyen d'une interface conviviale, directement contrôlable par un opérateur.

11. L'algorithme de lissage de lignes brisées a été intégré dans le graphique afin d'en visualiser le résultat avant de requérir l'exécution.

1.5 Conclusion

G^{en}oM nous a permis de développer une couche fonctionnelle complète qui offre toutes les capacités nécessaires au robot pour naviguer en environnement d'intérieur: acquisition de données capteurs, recalage du robot par rapport à l'environnement et affinement du modèle de celui-ci, adjonction d'obstacles ou d'amers découverts par le robot, planification de chemins, exécution de trajectoires avec évitement local d'obstacles, mise en place de surveillances, *etc.*.

Des algorithmes de structures très diverses (périodicité, temps de réponse, mode de terminaison), conçus par différentes personnes, ont pu être intégrés de façon homogène et cohérente au sein de la couche fonctionnelle. La logique interne des modules et les mécanismes d'activation des traitements et de transfert de données sont garantis par la génération automatique économisant ainsi de nombreux tests de mise au point. La génération automatique produit non seulement des modules dont on peut aisément appréhender et observer les activités, mais également des bibliothèques d'interactions standard dont les fonctions vont composer les *instructions élémentaires de programmation* du robot.

La couche fonctionnelle propose des services de base (surveillance sur des données proximétriques, calcul de chemins, segmentation d'images, ...) sur lesquels sont bâtis des services plus élaborés intégrés dans des modules clients des modules de base (*e.g.* l'évitement local d'obstacles utilise des activités des modules LOCO et PILO), offrant ainsi à l'opérateur ou au superviseur des comportements prédéfinis évolués. D'autres modules, intégrant d'autres comportements, pourront être ajoutés à la couche fonctionnelle (*e.g.* module pour garer le robot en environnement très contraint).

Ces services intègrent le premier niveau de réaction: les actions réflexes associées à des activités de surveillance ou de filtrage, et les procédures spécifiques de terminaison des activités lors d'exécution non nominale (états **fail** et **INTER**) qui participent grandement à la robustesse du système.

Les services de base et les services plus élaborés sont mis en œuvre par les niveaux décisionnels pour programmer des tâches complexes. La gestion de ces tâches nécessite en effet des informations globales concernant l'état du robot, l'environnement et la mission, ce dont bien sûr ne disposent pas les modules. L'état du robot est constamment observable au moyen des posters, et surtout l'évolution de la tâche est cadencée au rythme des répliques des services. Les bilans retournés par ces répliques jouent un rôle très important car ils informent immédiatement de l'état du système et permettent de décider de la façon dont la tâche doit être poursuivie.

La déclaration exhaustive, par le concepteur du module, des cas de défaillances ou d'exécutions non nominales des traitements mis en place (il est le plus à même d'en établir la liste), permet en effet de préparer au niveau du superviseur et/ou au niveau du planificateur (selon le degré d'urgence) des réactions adéquates et/ou des stratégies de remplacement (échec de localisation, échec de planification de chemins, échec du contournement local d'obstacles, panne d'un capteur, ...).

Un premier exemple de contrôle du robot par un opérateur au moyen de l'interface graphique $\mathcal{G}r\mathcal{H}2$ a été présenté. Cette couche fonctionnelle a été intégrée au sein d'architectures de contrôle dans le cadre d'expérimentations où le robot exécute des missions en coordonnant ses activités de façon autonome: deux exemples démonstratifs développés dans le cadre des projets EDEN et STRADA seront présentés dans les chapitres 3 et 4.

Chapitre 2

Développement de fonctions pour le déplacement et la localisation

Dans le cadre du développement de la couche fonctionnelle d'*Hilare2* nous avons conçu et intégré des fonctions de base pour un robot mobile qui ont trait à la locomotion et à la localisation du véhicule. Ces fonctions, que nous présentons maintenant, sont les algorithmes des codels des modules LOCO, PILO et LOCEXT. Elles proposent des concepts relativement généraux, et pour certaines originaux, applicables à d'autres véhicules. Elles sont une bonne illustration de types d'algorithmes qui peuvent être intégrés dans des modules.

Nous allons d'abord présenter les fonctions de déplacement pour véhicule non-holonome en traitant successivement de deux problèmes spécifiques: l'asservissement en position et l'exécution de trajectoires; puis nous exposerons une technique de localisation externe par caméras.

2.1 Contrôle des mouvements de véhicule non-holonome

La difficulté particulière que pose le contrôle de mouvement de véhicules provient du fait que, pour des raisons de simplicité et d'efficacité de la partie mécanique, la majorité d'entre eux ne disposent que de deux degrés de liberté pour évoluer dans un espace de configurations de degré trois (x, y, θ): ces véhicules sont dits *non-holonomes*. Pour la voiture, les degrés de liberté sont la direction et l'accélération, et pour *Hilare2*, l'accélération sur chacune des roues. La non-holonomie se traduit physiquement par le fait que ces véhicules ne sont pas omni-directionels, en particulier les véhicules cités ne peuvent se translater latéralement. Notons cependant que les robots de la famille *Hilare*, contrairement à la voiture, n'ont pas de borne minimale sur le rayon de giration et peuvent donc pivoter sur place.

La non-holonomie pose un triple problème pour le contrôle de mouvement en environnements contraints:

1. La détermination de chemins sans collision.
2. La génération de trajectoires[1] qui respectent la cinématique et les limitations dynamiques du véhicule.
3. L'asservissement par retour d'état d'un système non-holonome.

1. Selon la dénomination employée ici, un *chemin* caractérise uniquement la géométrie alors qu'une *trajectoire* exprime également la dynamique de parcours (vitesses et accélérations).

La détermination de chemins sans collision, qui a fait l'objet de nombreuses études, n'est pas abordée ici. Un planificateur sera présenté dans le chapitre 4. Le chemin géométrique peut d'ailleurs être défini par un opérateur, par exemple au moyen de l'interface graphique $\mathcal{G}r\mathcal{H}2$. Les points 2 et 3 sont la raison d'être des modules PILO et LOCO dont les algorithmes seront décrits dans les deux sections suivantes.

Le modèle cinématique d'un véhicule à deux roues motrices est visualisé sur la figure 2.1: le point de référence (x, y) est situé au milieu de l'entraxe des roues de longueur L. L'orientation du véhicule est notée θ. v_d et v_g sont, respectivement, les vitesses linéaires de la roue droite et de la roue gauche. Les entrées du système commandé sont les accélérations droite et gauche u_d et u_g. Les vitesses et les accélérations sont bornées: $|u_d| \leq a$ et $|u_g| \leq a$.

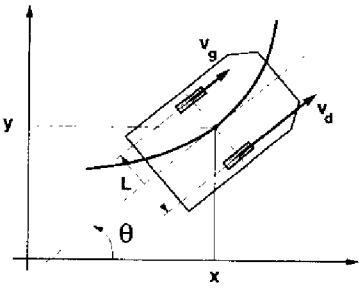


FIG. 2.1 – *Le modèle cinématique d'*Hilare2

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v_d} \\ \dot{v_g} \end{pmatrix} = \begin{pmatrix} \frac{1}{2}(v_d + v_g)\cos\theta \\ \frac{1}{2}(v_d + v_g)\sin\theta \\ \frac{1}{L}(v_d - v_g) \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} u_d + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} u_g$$

2.1.1 Asservissement d'un véhicule non-holonome

Il a été démontré qu'il n'existait pas, pour des véhicules ayant la cinématique d'*Hilare2*, de commande par retour d'état pure continue permettant d'asservir le robot en position et en orientation (voir [Samson 90]).

Différentes alternatives ont été envisagées: [Canudas de Wit 91] propose une loi de commande continue par morceau alors que [Samson 91] et [Kanayama 91] analysent une loi de commande continue mais instationnaire. Ces solutions permettent, de façon *exclusive*, soit de stabiliser le robot autour d'une configuration quelconque (x, y, θ) mais sans maîtrise du mouvement, soit de le réguler sur une trajectoire matérialisée par une consigne strictement *instationnaire*. Les limitations de ces solutions apparaissent d'elles-mêmes.

Par contre, on peut montrer ([Samson 90]) qu'en ne contrôlant directement que deux des trois paramètres: la position (x, y), il existe des lois d'asservissement par retour d'état avec une convergence exponentielle vers cette position. Qui plus est, à la condition de ne pas asservir un point sur l'axe des roues motrices, elles garantissent également une convergence en θ vers une constante qui, lors de l'exécution d'une trajectoire qui satisfasse les contraintes cinématiques, est la tangente à cette trajectoire.

La loi de commande que nous avons retenue intègre en outre des termes intégrateurs et des termes d'anticipation. Ces derniers permettent, dans le cas de consignes instationnaires, d'annuler le traînage lors qu'il est prédictible. Les consignes d'asservissement comporteront donc, outre la position, les vitesses instantanées linéaire et angulaire souhaitées en ce point. Ces vitesses sont l'expression de la trajectoire nominale qui est régulée en fonction de la consigne en position.

On trouvera dans l'annexe C la loi de commande et la démonstration de ses propriétés de convergence.

Cette loi de commande a été intégrée dans le codel `odoAndAsserv` de l'activité d'asservissement du module LOCO (voir §1.2.1.2). Si l'on souhaite asservir le robot en un point fixe donné, la consigne sera transmise par la requête de contrôle `locoGoTo` avec des vitesses nulles. Dans le cadre d'un suivi de trajectoire ou de point (ce qui est équivalent) la consigne complète (*i.e.* position et vitesses) est récupérée par l'activité de suivi de consigne `locoTrack` depuis le poster spécifié en paramètre de la requête d'exécution correspondante.

La détermination du vecteur d'état (position et vitesses instantanées) qui intervient dans l'asservissement et qui est également calculé par le codel `odoAndAsserv`, est obtenu par l'intermédiaire de codeurs optiques disposés sur les roues odométriques. En comptabilisant les incréments sur chaque roue on déduit du modèle cinématique les vitesses linéaire et angulaire instantanées du véhicule, et donc les déplacements curviligne et angulaire élémentaires $(ds, d\theta)$. La configuration (x, y, θ) du robot est alors obtenue en projetant et en intégrant ces variations élémentaires dans un repère global. L'orientation θ peut également être directement fournie par un gyroscope. Un calcul probabiliste de l'erreur odométrique permet d'estimer l'incertitude sur la configuration. La position et son incertitude sont exportées dans le poster `locoRobot`.

La qualité de l'asservissement peut être visualisée sur la figure 2.2: la trajectoire de consigne (qui comporte un point de rebroussement sur anticlothoïde avec vitesse de rotation constante - voir la génération de trajectoires ci-dessous) et la trajectoire exécutée sont quasiment superposées.

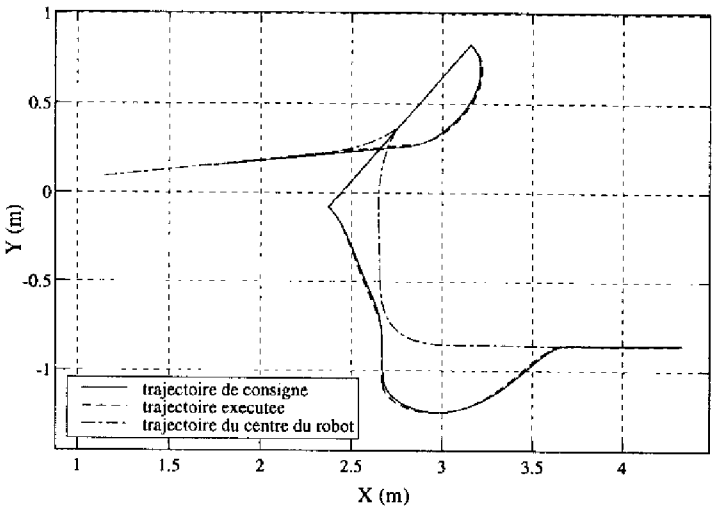


FIG. 2.2 – **Asservissement sur une trajectoire.** *La trajectoire exécutée par le centre du robot est en traits discontinus. L'écart maximal entre la trajectoire de consigne (trait plein) et la trajectoire exécutée (pointillés) est d'environ 1cm pour un point asservie situé à 60cm de l'axe des roues. (Véhicule de 400kg, accélération de $1m/s^2$ et $3rd/s^2$, vitesses de 0.5m/s et 1rd/s).*

La flexibilité offerte par G^{en}oM permet aisément de tester différents types d'asservissement par simple substitution du codel d'asservissement. Ainsi une loi proposée par Greg Walsh d'après [Teel 92] a été testée en simulation et d'autres lois en (x, y, θ) ou permettant le contrôle d'une remorque sont en cours de développement et devraient être testées prochainement.

2.1.2 Génération de trajectoires cinématiquement admissibles

Contrairement au problème d'asservissement de véhicules non-holonomes, celui de la génération de trajectoires a donné lieu à de nombreux travaux qui consistent à déterminer des trajectoires qui respectent la cinématique du véhicule. Selon les solutions proposées, les tra-

jectoires joignent directement des configurations (x, y, θ) ou lissent des chemins prédéfinis. En effet, la majorité de ces planificateurs géométriques produisent des chemins composés de lignes brisées ou de segments et d'arcs de cercles qui forcent l'arrêt du robot lors des discontinuités de courbure.

2.1.2.1 Etat de l'art

Les contraintes essentielles des trajectoires générées sont la continuité de la tangente et de la courbure. Une technique maintenant classique dans le domaine de la robotique fait appel aux clothoïdes, courbes dont la courbure varie linéairement avec l'abscisse curviligne. Cette méthode est utilisée depuis de très nombreuses années dans la conception d'autoroutes ou de voies de chemin de fer afin d'en joindre en douceur les portions linéaires et circulaires.[2] Cette solution avait déjà été adoptée sur nos robots *Hilare1* et *Junior* pour enchaîner des segments de droites [Chochon 83] ou des segments de droites et des arcs de cercles [Khoumsi 88].

De nombreux auteurs se proposent de produire directement une trajectoire qui joigne une séquence de configurations $\{(x, y, \theta)_i\}$ sans marquer d'arrêt. Ainsi, dans [Shin 90], Shin et Singh génèrent des trajectoires composées exclusivement de clothoïdes qui, de plus, intègrent la contrainte de limitation en rayon de giration.

Dans [Kanayama 89], Kanayama et Hartman joignent les configurations deux à deux par des spirales cubiques dont les rayons de courbure sont supérieurs à ceux qui seraient obtenus par des clothoïdes: les virages sont moins serrés. Un critère qui caractérise la "rondeur" de la trajectoire y est défini (une fonction quadratique de la courbure et de sa dérivée).

Delingette *et al.* [Delingette 91] ont généralisé cette méthode par les "splines intrinsèques" dont la courbure est une fonction polynomiale de l'abscisse curviligne (*i.e.* généralisation des clothoïdes et des spirales cubiques), en limitant le rayon de courbure et en introduisant des points de contrôle (similaires aux approches de CAO).

On peut encore citer de nombreux travaux tels que ceux de Segovia *et al.* [Segovia 91] qui font intervenir les courbes de Bezier, ou de [Nelson 89] qui définissent des courbes dont le rayon est une fonction polynomiale de l'angle polaire.

Bien que très intéressantes en termes de lissage de trajectoires, ces solutions se sont généralement trop écartées du problème de la planification en milieu contraint: elles ne considèrent pas les obstacles qui environnent le robot et qui ont rendu nécessaire la phase de planification (en l'absence d'obstacle, la ligne droite est une trajectoire tout à fait satisfaisante). La solution que nous proposons considère explicitement les écarts tolérables par rapport au chemin initial.

Un autre aspect qui n'a jamais été abordé à notre connaissance est la prise en compte des points de rebroussement qui, bien que rarement rencontrés sur autoroutes, peuvent se présenter lors de la planification de chemins en environnement très contraint (*e.g.* les créneaux). Nous nous proposons de traiter ces rebroussements au moyen de trajectoires baptisées *anti-clothoïdes*. Les anticlothoïdes, dont le nom sera justifié par la suite, sont des dévellopantes de cercle: le rayon de courbure varie proportionnellement à la tangente à la trajectoire.

2. En tournant continûment le volant sans faire varier la vitesse linéaire on exécute une clothoïde. Lorsque le rayon de giration souhaité est atteint il suffit alors de maintenir l'angle du volant pour suivre la partie circulaire du virage.

Enfin, les limitations de vitesse et d'accélération, qui sont souvent omises dans ces études seront considérées.

2.1.2.2 Le générateur de trajectoire d'*Hilare2*

Le générateur de trajectoires qui a été intégré dans le module PILO utilise quatre types de trajectoires primitives: les *segments*, les *arcs de cercle*, les *clothoïdes* et les *anticlothoïdes*. Comme nous allons le voir ces trajectoires sont localement optimales en temps et une commande élémentaire en accélération permet de les exécuter.

Nous allons dans un premier temps présenter la commande qui produit ces trajectoires primitives, puis après une analyse de ces trajectoires, nous proposerons deux algorithmes de lissage: le lissage de lignes brisées et le lissage de trajectoires de type Reeds & Shepp (composées de segments de droite et d'arcs de cercles). Ces deux fonctions sont disponibles sous la forme de requêtes d'exécution adressées au module PILO. Elles lissent également les points de rebroussement et permettent de maîtriser la déviation par rapport au chemin initial.

▷ **La commande "bang-bang" et les trajectoires primitives d'*Hilare2*** En appliquant le Principe du Maximum il a été prouvé (voir [Jacobs 91]) que la commande optimale en temps du système de la figure 2.1 page 102 vérifie: $|u_d| = |u_g| = a$. Il s'agit d'une commande "bang-bang" (commande pour laquelle les accélérations sont maximales).

Selon que u_d et u_g soient de signes opposés ou de même signe, les trajectoires optenues sont des clothoïdes ou des anticlothoïdes. Soulignons que ce résultat n'exprime qu'une condition nécessaire: on ne sait pas à l'heure actuelle déterminer la trajectoire optimale qui lie deux configurations.

- **Les clothoïdes** (et les rotations pures)
  Considérons la commande "bang-bang" qui vérifie: $u_d = -u_g = a$. L'accélération linéaire du véhicule est la moyenne des accélérations sur chaque roue: $\alpha = 0$. La vitesse linéaire est donc constante: $v = v_0$ et l'abscisse curviligne s'écrit: $s(t) = v_0 t + s_0$. L'accélération angulaire est la différence des accélérations sur chaque roue divisée par l'entraxe: $\gamma = 2a/L$. D'où la vitesse angulaire: $\omega(t) = \frac{2a}{L}t + \omega_0$. Finalement la courbure en fonction du temps s'écrit:

$$\kappa(t) = \frac{\omega(t)}{v(t)} = \frac{2a}{Lv_0}t + \frac{\omega_0}{v_0} = \frac{2a}{Lv_0^2}s(t) + \frac{\omega_0}{v_0} \tag{2.1}$$

  Cette équation est l'expression intrinsèque d'une clothoïde ($\kappa = k_c s$) dont la constante caractéristique k_c vaut $\frac{2a}{Lv_0^2}$. On trouvera l'équation cartésienne et le graphe d'une clothoïde dans l'annexe D.

  Remarques:
  - La vitesse linéaire de parcours de cette clothoïde est constante.
  - Pour $v_0 = 0$, la trajectoire est une rotation pure qui peut être considérée comme une clothoïde dégénérée dont la constante caractéristique est infinie.
  - L'équation (2.1) montre qu'une clothoïde permet de joindre de façon continue une courbe de courbure nulle (une droite) et une courbe de courbure non-nulle.

- **Les anticlothoïdes** (et les translations)
  Considérons maintenant le cas: $u_d = u_g = a$. L'accélération linéaire est alors constante: $\alpha = a$, et la vitesse linéaire vaut: $v = at + v_0$. L'accélération angulaire est nulle, donc

la vitesse angulaire est constante: $\omega = \omega_0$ et l'angle: $\theta(t) = \omega_0 t + \theta_0$. D'où le rayon de courbure en fonction du temps:

$$\rho(t) = \frac{v(t)}{\omega(t)} = \frac{a}{\omega_0}t + \frac{v_0}{\omega_0} = \frac{a}{\omega_0^2}\theta(t) + \frac{v_0}{\omega_0} \tag{2.2}$$

Cette équation est l'expression intrinsèque d'une développante de cercle ($\rho = k_a\theta$) dont la constante caractéristique k_a vaut $\frac{a}{\omega_0^2}$. Les propriétés duales de cette courbe par rapport aux clothoïdes nous ont amené à la rebaptiser *anticlothoïde* (voir annexe D). On trouvera l'équation cartésienne et le graphe d'une anticlothoïde dans l'annexe D. Ainsi on remarque que:

- La vitesse angulaire de parcours de cette anticlothoïde est constante.
- Pour $w_0 = 0$, la trajectoire est une translation qui peut être considérée comme une anticlothoïde dégénérée dont la constante caractéristique est infinie.
- L'équation (2.2) montre qu'une anticlothoïde permet de joindre de façon continue une courbe de rayon de courbure nul (une rotation pure) et une courbe de courbure non-nulle.

On remarquera que le point origine $(0, 0)$ de l'anticlothoïde est un point de rebroussement de rayon de giration nul (rotation pure) qui marque un changement dans le sens de translation.

Il est intéressant de noter que dans le cadre d'études de variateurs de commande à basse consommation pour véhicule à deux roues motrices ([Bouscayrol 95]), les accélérations imposées sur les deux roues sont de même amplitude. Les trajectoires ainsi générées peuvent être approximées par morceau par des clothoïdes et par des anticlothoïdes.

▷ **De l'usage des clothoïdes et des anticlothoïdes** Chaque mouvement commence et se conclut par des vitesses nulles ($v = \omega = 0$). En appliquant une commande optimale sur les roues on ne peut donc quitter ou rejoindre une position d'arrêt que par une translation ou une rotation pure. D'après ce qui précède, la translation peut alors s'enchaîner par une clothoïde, et la rotation par une anticlothoïde. La figure (2.3 page suivante) présente l'ensemble des enchaînements possibles. Les types de trajectoire sont les nœuds du graphe, alors que les conditions de transition en sont les arcs. Le cercle qui sera introduit lors du lissage des trajectoires de Reeds & Shepp peut s'enchaîner indifféremment avec les clothoïdes et les anticlothoïdes non dégénérées ou encore depuis la position d'arrêt. Les cercles ne peuvent être produits par une commande "bang-bang" car le rapport des accélérations est dans ce cas égal au rayon souhaité.

Ces cinq types de trajectoires sont obtenus par des accélérations constantes par morceau et donc des profils de vitesses trapézoïdaux. La figure 2.4 page ci-contre montre un exemple d'enchaînement de trajectoires et les profils de vitesses linéaire et angulaire, obtenus à l'issue de l'exécution de cette trajectoire et visualisés à l'aide de `StethoScope`. La commande de ces trajectoires est donc extrêmement simple et bien adaptée au contrôle d'un système temps réel.

Le module PILO peut exécuter toute trajectoire composée de ces primitives. Un codel procède au calcul des *profils de vitesse optimaux* de la trajectoire complète en appliquant les relations: $k_a = \frac{a}{\omega_0^2}$ et $k_c = \frac{2a}{Lv_0^2}$, et en respectant les limitations en vitesse et accélération linéaire et angulaire. Les vitesses nulles imposées en fin de trajectoire peuvent nécessiter un

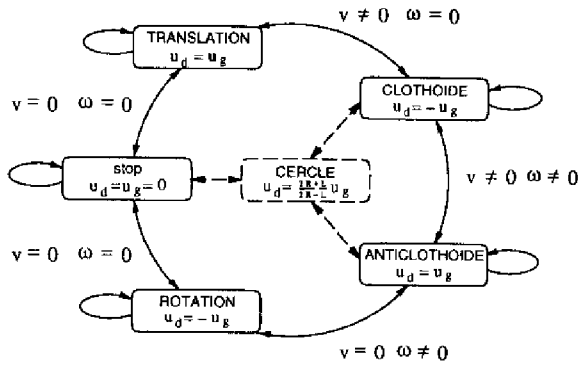
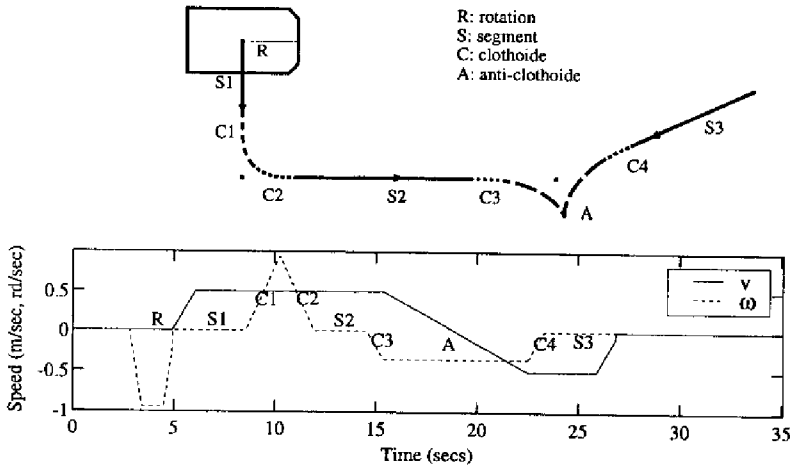


FIG. 2.3 – *Enchaînement des clothoïdes et des anticlothoïdes.*

FIG. 2.4 – *Exemple de trajectoire combinant clothoïdes et anticlothoïdes et les profils de vitesse correspondants.*



calcul par retour arrière le long de la trajectoire.

Les consignes en vitesse et position instantanée sont alors calculées périodiquement et exportées dans un poster. Ce calcul en ligne permet à tout instant de modifier les vitesses de parcours grâce à la requête de contrôle `piloSlow`.[3] On peut en particulier arrêter le véhicule sur sa trajectoire et repartir.

▷ **Application au lissage de lignes brisées** La majorité des planificateurs géométriques produisent des lignes brisées. Les lignes brisées sont également un moyen aisé pour l'opérateur de spécifier un chemin. Cependant, exécutées telles quelles, elles généreraient des séquences de mouvements saccadés: [stop-...-TRANSLATION-stop-ROTATION-stop-TRANSLATION-stop-ROTATION-...-stop].

La requête `piloSmooth` du module PILO, qui prend en argument une ligne brisée, détermine une trajectoire de façon à:

- obtenir un mouvement lissé et sans aucun arrêt: les vitesses des roues seront continues et jamais simultanément nulles sauf aux points de départ et d'arrivée;
- ne pas s'écarter du chemin initial au delà d'un seuil fixé;
- respecter les sens de translation et de rotation imposés.

3. Pour rester sur une trajectoire donnée, il suffit de conserver le rayon de giration instantané: $\rho = v/\omega$ tout en respectant les limitations en accélération. Les vitesses seront bien sûr inférieures ou égales à celles du profil optimal: le paramètre de la requête `piloSlow` est un pourcentage de la vitesse linéaire maximale.

Les entrées du problème sont illustrées par le premier schéma de la figure (2.5). Un virage se compose de deux segments S_1 et S_2 formant un angle β. On suppose que le robot se présente sur le segment S_1 en marche avant (la marche arrière se déduit par symétrie). σ est le sens du mouvement sur le segment S_2, imposé par le planificateur: si $\sigma > 0$ alors S_2 est parcouru en marche avant, sinon S_2 est parcouru en marche arrière. δ exprime le sens de rotation: si $\delta > 0$ alors le virage est exécuté selon le sens trigonométrique, sinon il est exécuté selon le sens des aiguilles d'une montre. ε est l'écart maximal (à orientation égale du véhicule) de la trajectoire lissée par rapport à la ligne brisée. L'écart le plus grand se situant au sommet du virage, le couloir de la figure (2.5) représente la zone dans laquelle la trajectoire devra être contenue. Le virage produit par le lissage est symétrique par rapport à la bissectrice $\widehat{S_1S_2}$.

Selon les signes de σ et de δ, quatre cas sont donc à considérer. Le virage démarrant et se concluant par une courbure nulle, les portions finale et initiale seront nécessairement des arcs de clothoïde. Lorsque $\sigma < 0$, la trajectoire comporte un point de rebroussement qui ne peut être lissé que par une anticlothoïde.

Les trajectoires obtenues pour ces quatre cas sont représentées sur la figure 2.5. On trouvera dans [Fleury 95b] une étude détaillée et les solutions du problème.

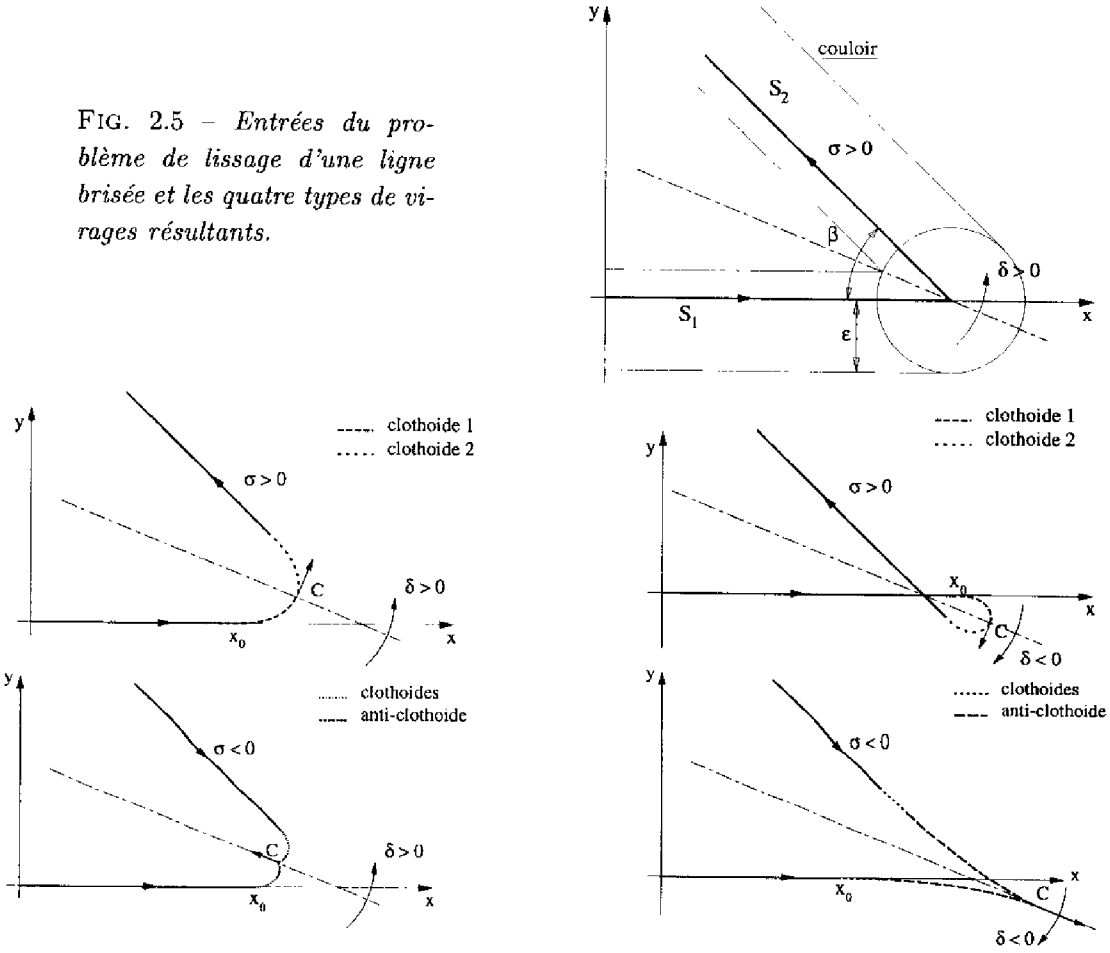


FIG. 2.5 – *Entrées du problème de lissage d'une ligne brisée et les quatre types de virages résultants.*

Le calcul de la géométrie de la trajectoire lissée est indépendant du calcul de sa dynamique et est effectué préalablement à son exécution. Le temps de calcul d'un virage sans rebroussement est de 0.3ms sur un processeur 68040 à 25MHz, et celui d'un virage avec rebroussement est de 5ms (sur le chronogramme de la figure 1.11 page 93 la phase **start** correspond au calcul d'une ligne brisée composée d'une dizaine de points de rebroussement). Ces temps sont suffisamment courts pour envisager de paralléliser le calcul géométrique et l'exécution, l'intérêt étant de concaténer dynamiquement des séquences de trajectoires sans marquer d'arrêt. L'activité serait alors scindée en une activité de calcul géométrique et une activité d'exécution de trajectoire (calcul et exportation des consignes) qui pourraient être invoquées parallèlement. Le temps de calcul de la consigne à chaque période est négligeable.

L'interface graphique $\mathcal{GrH}2$ autorise la saisie de lignes brisées. Le codel de calcul de la géométrie du virage lissé a également été intégré dans $\mathcal{GrH}2$ permettant ainsi de pré-visualiser les trajectoires qui vont être exécutées. La figure 2.6 qui présente une ligne brisée orientée et le résultat de son lissage a été obtenue par cette interface graphique (les annotations portées sur la ligne brisée ont été ajoutées après coup).

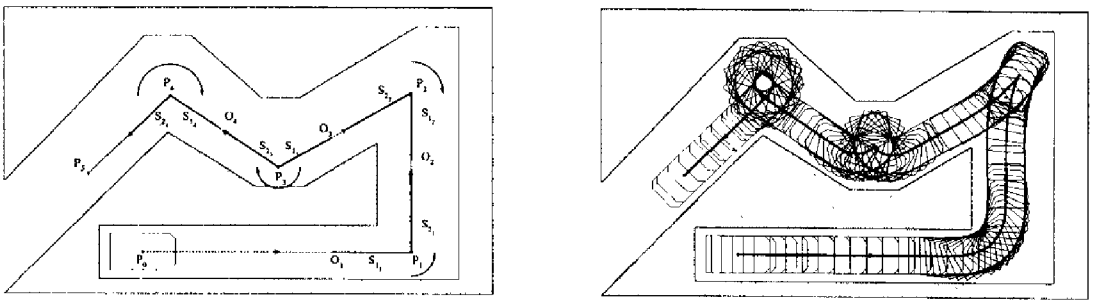


FIG. 2.6 – *Lissage d'une ligne brisée orientée.*

▷ **Application au lissage de chemins de Reeds & Shepp** Un planificateur de chemins sans collision qui intègre la limite du rayon de giration borné a été intégré dans un module nommé MP. Ce module a été utilisé sur les robots dans le cadre du projet STRADA qui sera présenté dans le chapitre 4. Le planificateur produit des chemins de type Reeds & Shepp qui sont des séquences particulières de segments de droites et d'arcs de cercles. Ces enchaînements introduisent des discontinuités de courbure qui doivent être lissées. L'algorithme de lissage qui est exposé dans l'annexe D a été intégré dans le module PILO. La requête `piloReedsShepp` permet de lisser la trajectoire puis de l'exécuter selon la procédure commune (le codel de la phase **exec** qui produit la consigne est commun aux différentes requêtes d'exécution de trajectoire).

2.2 La localisation externe

La localisation d'un robot par des capteurs proprioceptifs (odométrie, gyroscope) autorise une mise à jour en temps réel qui peut s'intégrer dans une boucle d'asservissement. Par contre, elle présente le double défaut de cumuler les erreurs au cours des déplacements et de ne pas se référencer à un repère absolu. Des techniques complémentaires de localisation extéroceptive devront donc être appliquées pour recaler le véhicule. L'environnement d'un robot mobile autonome n'étant pas a priori instrumenté, le robot doit être équipé de capteurs extéroceptifs embarqués. Cependant, une localisation externe absolue peut être envisagée dans certaines applications industrielles par exemple, et surtout elle présente l'intérêt, dans les phases de développement, de valider et de calibrer les techniques de localisation proprioceptives et extéroceptives embarquées ([Fleury 92, Fleury 93]). Une application à la calibration de l'odométrie sera présentée dans l'annexe E.

2.2.1 Le problème

Afin de disposer d'un moyen efficace pour localiser de façon absolue un ou plusieurs véhicules en mouvement sans avoir à structurer l'environnement, des caméras N&B ont été placées dans l'environnement de façon à couvrir la zone d'évolution des robots.

La localisation dynamique d'un objet mobile requiert des primitives perceptuelles très simples à extraire et à identifier: un motif de points matérialisés par des diodes infrarouges est disposé sur chaque robot.

Le problème, illustré par la figure 2.7, consiste à identifier et localiser le motif dans une image de luminance segmentée en points et d'en déduire la position du robot.

Les notations de la figure 2.7 sont les suivantes: $T_{A/B}$ est une transformée qui exprime la position d'un repère A par rapport à un repère B. Avec les repères: R: robot; C: caméra; I: image; A: repère absolu (de référence) et M: motif de points.

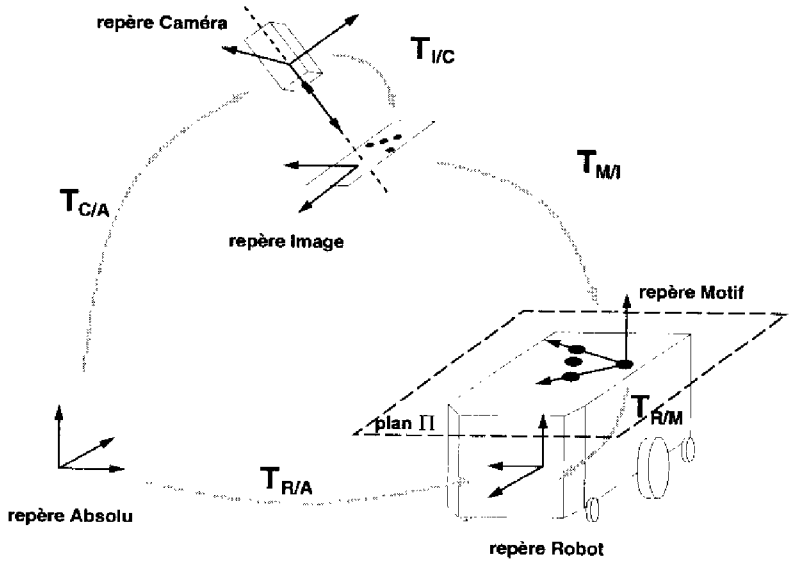


FIG. 2.7 – *Le graphe des repères.*

Il s'agit donc de déterminer la transformée $T_{R/A}$ qui exprime la position du robot R dans

le repère de référence A. D'après la figure 2.7 page ci-contre, cette transformée s'écrit:

$$T_{R/A} = T_{C/A} * T_{I/C} * T_{M/I} * T_{R/M}$$

Les trois transformations $T_{C/A}, T_{I/C}$ et $T_{R/M}$ sont constantes. Elles correspondent respectivement à la calibration de la caméra par rapport à l'environnement (les paramètres extrinsèques de la caméra), à la calibration des paramètres intrinsèques de la caméra et à la calibration de la position du motif par rapport au repère robot. Ces calibrations sont décrites dans l'annexe E; en particulier, une calibration automatique par moindres carrés de la position du motif grâce à la localisation externe y est proposée. On montre en effet qu'il suffit de connaître les mouvements *relatifs* du robot d'une part et du motif d'autre part pour procéder à cette calibration. Nous supposerons par la suite que ces transformées sont connues.

2.2.2 Identification et localisation du motif

Le problème de la localisation d'un objet décrit par des points caractéristiques a déjà été abordé par de nombreux auteurs ([Horaud 89, Rives 83]). Une solution maintenant classique [Faugeras 86, Mohr 88] consiste à considérer tous les appariements entre les points de l'objet et les points de l'image. A chaque quadruplet d'appariements correspond une position 3D de l'objet qui sera validée ou invalidée selon que d'autres appariements satisfont ou non cette hypothèse. La combinatoire de cette technique interdit des performances temps réel, d'autant que dans le cas considéré l'image peut être bruitée par des points parasites et certains points du motif peuvent être occultés.

Sans remettre en cause le principe de la boucle de prédiction/vérification, cette technique sera très sensiblement améliorée: a) en se ramenant temporairement à un problème 2D, b) en procédant à un filtrage rapide pour prédéterminer les appariements les plus probables et supprimer les artefacts les plus flagrants. Les appariements présélectionnés permettront d'initialiser la boucle d'identification/localisation.

▷ **Un problème 2D** Les motifs de points considérés sont coplanaires et nous allons supposer dans un premier temps, pour la procédure d'identification, que le mouvement du robot entre deux itérations est plan. L'intérêt est de disposer ainsi d'une estimée du plan Π du motif (voir la figure 2.7 page précédente). Le problème s'en trouvera considérablement simplifié car: a) dans un plan deux points identifiés suffisent à la localisation de l'objet, ce qui réduit considérablement la combinatoire; b) après une projection perspective inverse des points de l'image dans le plan Π, nous obtenons une bonne approximation des positions vraies des diodes. Ces points de la scène pourront alors être directement comparés à ceux du modèle et être ainsi préclassifiés afin de guider la procédure d'identification.

▷ **Sélection des appariements les plus probables** Nous proposons une méthode efficace pour opérer un premier tri. Elle permet de supprimer les nuages de points parasites ainsi que les points isolés, et surtout de présélectionner les appariements les plus probables.

Pour cela on va considérer des caractéristiques géométriques de motifs de points regroupés en primitives perceptuelles ou percepts. Les percepts les plus simples dans une image de points sont les bi-points et leur unique caractéristique est la longueur.[4]

4. A partir de trois points on peut disposer de caractéristiques beaucoup plus discriminantes telles que le rapport des distances, les angles, ..., cependant la combinatoire les rendrait trop coûteuses en temps et seul un filtrage grossier est nécessaire.

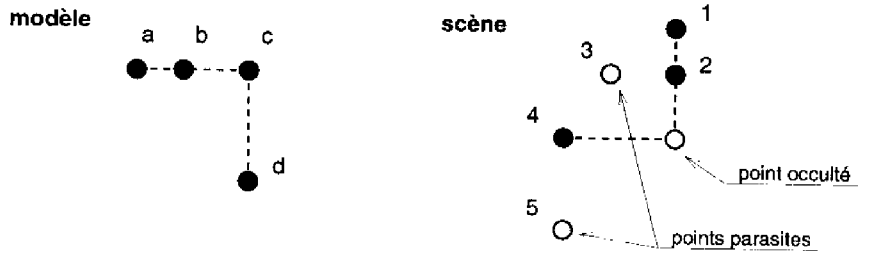
On procède d'abord à une classification des percepts: chaque bi-point du modèle M_{ij}, composé des points m_i et m_j, constitue une classe caractérisée par sa longueur l_{ij} entre les deux points. Si un percept S_{kl}, composé des points s_k et s_l de la scène, a une longueur incluse dans l'intervalle $[l-\varepsilon, l+\varepsilon]$, où ε est l'incertitude de mesure de distance entre deux points de la scène, alors le percept S_{kl} appartient à la classe M_{ij} ($S_{kl} \in M_{ij}$).

On déduit de cette classification les appariements probables de points: si le percept S_{kl} appartient à la classe M_{ij}, alors on peut supposer que s_k (ou s_l) est l'image du point m_i ou m_j. On définit un score d'appariement entre deux points $score(s_k, m_i)$ comme étant le nombre de classes M_{ij} pour lesquelles il existe un point s_l tel que $S_{kl} \in M_{ij}$. Un percept qui n'appartient à aucune classe est noté $S_{kl} \in \emptyset$. Le score maximal vaut $N(N-1)/2$, N étant le nombre de points du modèle.

La limite de cet algorithme est qu'un percept peut appartenir à plusieurs classes (ce qui n'a pas de sens physiquement) et augmenter d'autant les scores d'appariements de ces points. Cependant, la longueur étant notre critère d'identification, le modèle doit être défini de telle sorte que ce phénomène ne se produise pas trop fréquemment, afin que les valeurs des scores en soient peu affectées.

Afin d'éliminer les nuages de points on comptabilise également le nombre total de points s_l qui font que S_{kl} appartient à une classe. Un nombre anormalement élevé caractérise clairement un agrégat de points.

Cet algorithme, contrairement aux méthodes exhaustives, a une complexité polynomiale. Les appariements de plus fort score sont sélectionnés pour initialiser la boucle de prédiction/vérification. Dans l'exemple de la figure 2.2.2 le choix se fera parmi les couples $(b,2)$, $(a,1)$ ou (d,4) et le point 5 peut être éliminé.



Classification des percepts

	classes						
$\in$	ab	bc	cd	da	ac	bd	$\emptyset$
pe	12	23		14		24	13
rc		34		35			15
cp							25
ts							25

Score d'appariements

	a	b	c	d	$\emptyset$
1	2	1	0	1	2
2	1	3	1	1	1
3	1	1	1	1	2
4	1	1	0	2	2
5	1	0	0	1	3

FIG. 2.8 – *Exemple d'identification.*

▷ **La boucle d'identification/localisation** A partir de deux appariements, on déduit une position possible $T_{M/\Pi}$ pour le motif dans le plan Π. Il est alors aisé de déterminer si d'autres points du modèle peuvent être appariés à un point de la scène en appliquant la relation $P_\Pi = T_{M/\Pi} P_{/M}$ (à l'incertitude ε près). Au fur et à mesure que de nouveaux

appariements sont identifiés, la position est incrémentalement affinée par une technique de moindres carrés décrite dans l'annexe E. A partir de quatre ou cinq points on considère que le motif est localisé. En cas d'échec, on sélectionne une autre paire d'appariements parmi les plus probables. Le synopsis de la figure 2.9 résume l'algorithme de localisation.

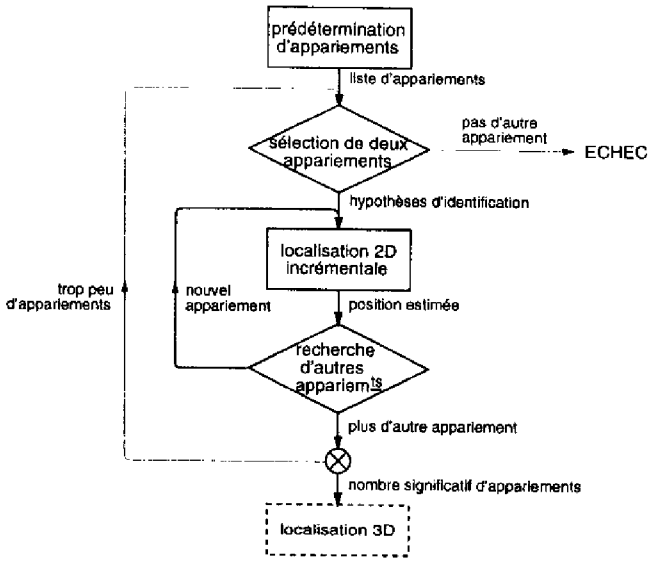


FIG. 2.9 – *Synopsis de la procédure de localisation.*

▷ **La localisation du robot** La localisation du robot se déduit directement de celle du motif et de la transformation constante $T_{M/R}$ qui lie les deux repères. Si le robot, et donc le motif, évolue dans un plan horizontal, alors le plan Π est invariant et la localisation 2D suffit à caractériser la position du robot.

Dans le cas contraire, il faudra résoudre le problème classique de la perspective à n-points (voir [Horaud 89, Tsai 87]).

2.2.3 Intégration et résultats

La procédure a été intégrée dans une boucle de suivi d'objet qui consiste à estimer la direction et la vitesse du mouvement du véhicule et donc la position du motif à l'itération suivante. Cette information permet de sélectionner la caméra adéquate et de réduire la fenêtre de recherche dans l'image.[5]

L'installation comporte trois caméras dont les champs se chevauchent et couvrent une zone d'environ $5 \times 10\text{m}^2$. La figure 2.10 page suivante est le résultat d'une expérimentation visualisée par $\mathcal{G}r\mathcal{H}2$ qui accède aux posters de position des modules LOCO et LOCEXT. On y distingue les trois champs des caméras. La trajectoire de consigne apparaît en trait plein, les positions calculées par l'odométrie sont en trait plein et celles obtenues par la localisation externe sont en pointillé. La continuité des trajectoires illustre bien la qualité de la calibration des caméras dans un repère commun. Les motifs utilisés se composent de sept ou huit points circonscrits à un rectangle d'environ $40 \times 50\text{cm}^2$. Rappelons que quatre points suffisent, la redondance permettant de parer à des occultations partielles et d'améliorer l'estimation de

5. Plus la fenêtre est petite, plus la segmentation de l'image sera rapide, et moins il y aura de points parasites.

la position.

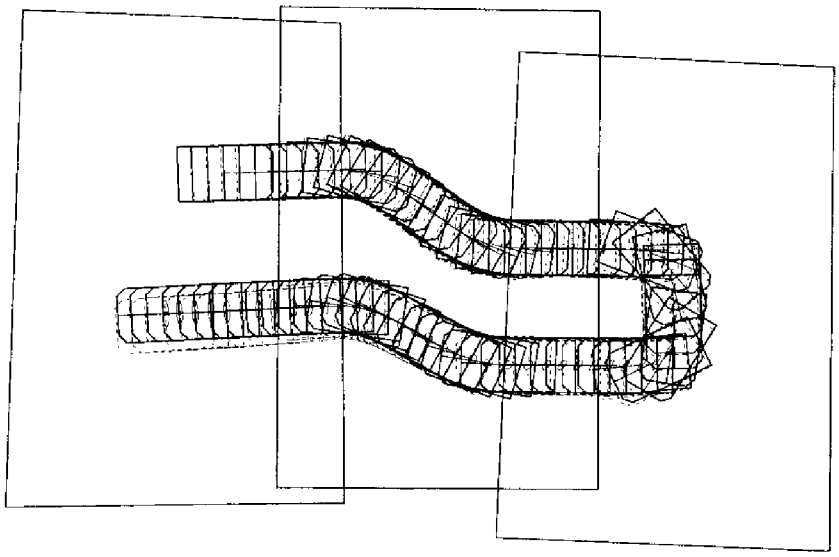


FIG. 2.10 – *La trajectoire de consigne et les positions du véhicule selon l'odométrie (sans utilisation du gyroscope) - en traits continus, et selon la caméra - en pointillés.*

Les performances du système sont résumées dans le tableau ci-dessous:

Discrétisation	Caméras de distance focale de 8mm à 6m du sol: **9.5 × 6.0 mm^2 par pixel**
Répétabilité	Dispersion moyenne du motif après 1500 localisations **X = 1 mm, Y = 2 mm, θ = 0.4 deg**
Précision	Mire calibrée de 1 × 2.8m^2 composée de 84 points: Dispersion **Moy: 2 mm, Max: 1 cm, Ecart type: 2mm**
Temps de calcul	Processeur 68030 (25MHz). Fenêtre de 150 × 150 pixels. 8 points sans parasite: **240 ms**
Robustesse	1. Autorise la présence de nombreux **artefacts** (jusqu'à 150) 2. Autorise des **occultations** partielles du motif.

Les résultats donnés ci-dessus ont été obtenus a) sans appliquer la correction de la distorsion de l'image (voir l'annexe E), ce qui améliorerait encore la précision de la localisation et b) sans optimisation du "tracking" (au lieu d'estimer la position du motif à l'itération suivante on pourrait estimer directement celles des points), ce qui guiderait et accélèrerait la phase d'appariement.

L'image vidéo commentée de la figure 2.11 page suivante illustre bien la robustesse du système: le motif a pu être identifié dans cette image particulièrement bruitée grâce au filtrage des nuages de points parasites. Les croix sont les points retenus à l'issu de la segmentation.

Cet algorithme a été intégré dans le module LOCEXT. La requête d'exécution `locextFind` procède à la localisation et retourne la position avec la réplique. La requête de suivi `locextTrack` démarre le processus de tracking. La position est alors exportée dans un poster à la fréquence du tracking. Des requêtes de contrôle permettent de modifier les gains des caméras, d'enre-

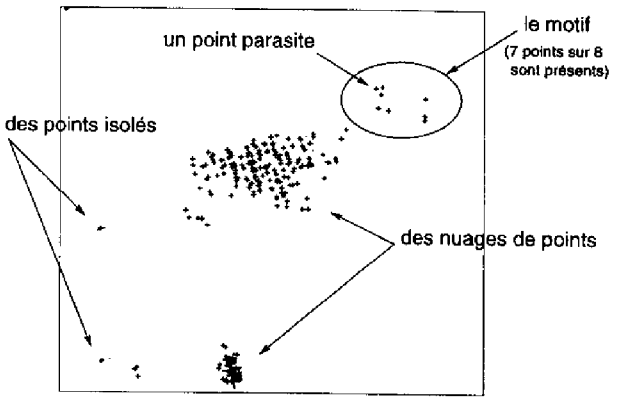


FIG. 2.11 – *Une image réelle. Les nuages de points sont dûs à des reflets (infra-rouges) du soleil sur le sol.*

gistrer ou de redéfinir des motifs ou de sélectionner le motif qui doit être suivi.

2.3 Conclusion

Nous avons développé des fonctions de base indispensables à la commande de robots mobiles:

- la localisation proprioceptive (odométrie et gyroscope),
- l'asservissement en position,
- la génération et l'exécution de trajectoires non-holonomes.

La fonction de localisation absolue permet de calibrer les procédures de localisation embarquées (E page 170), et est employée à la localisation des robots qui ne sont pas équipés de méthode de localisation relative à l'environnement. En particulier lors d'application multi-robots, il est crucial de valider et contrôler les localisations produites par les robots. On notera que cette technique permet également de localiser dans un repère absolu des objets autres que le robots, par exemple les objets fixes de l'environnement[6].

Ces fonctions composent les codels des modules LOCO, PILO et LOCEXT et se présentent ainsi sous la forme de services programmables dont peuvent disposer les autres modules ou le superviseur d'exécution par l'intermédiaire des requêtes. Ainsi, l'asservissement en position intervient dans le suivi de trajectoires, mais aussi lors de suivi visuel ou d'évitement d'obstacles. Le module PILO peut être interfacé avec des planificateurs de chemins très divers.

Le caractère générique des algorithmes élaborés, associé au contexte d'intégration qu'offre G^{en}$_{o}$M, ont permis leur intégration sur différents robots.

6. On dispose à cet effet d'une plaque amovible de diodes

Chapitre 3

EDEN: Une application en robotique d'extérieur

Les applications robotiques en environnements naturels non-structurés telles que l'exploration de sites planétaires ou sous-marins ou l'intervention après catastrophes sont typiquement des situations qui requièrent un niveau d'autonomie important. En effet, le robot doit opérer dans un environnement pour lequel on ne dispose pas d'informations a priori qui devront être acquises par l'intermédiaire de ses capteurs. La limitation des interactions entre la station opérateur et le robot, caractéristique de ce type d'application, rend nécessaire des capacités embarquées pour analyser ces données et déterminer la stratégie à adopter pour accomplir la mission [Giralt 93].

L'Expérimentation de Déplacement en Environnement Naturel (EDEN) se place dans ce contexte et consiste à faire exécuter au robot ADAM[1] la mission canonique de navigation autonome **"Aller à (but)"** [Chatila 93b, Lacroix 94]. Ce projet se situe dans la continuité des projets AMR [Lacombe 91] et VAP[2] [Giralt 92].

Cette expérimentation, présentée dans ce chapitre, permet également de montrer la pertinence de l'organisation modulaire des multiples fonctions de déplacement, de perception, de modélisation et de planification de mouvement de la couche fonctionnelle, ainsi que son intégration dans une architecture de contrôle complète selon les principes et choix architecturaux énoncés dans la première partie de ce mémoire.

Bien que cette expérimentation ait été mise en place avant que $G^{en}{}_{o}M$ soit disponible, la structure et le comportement des modules, ainsi que les modes d'interaction entre eux et avec le niveau décisionnel avaient été préalablement spécifiés et ont été respectés lors de l'intégration. Les leçons tirées de cette implémentation ont permis la mise au point de $G^{en}{}_{o}M$.

Après la présentation du robot ADAM, nous verrons les problèmes particuliers que pose la navigation autonome en environnement naturel et les solutions qui ont été adoptées et développées dans les thèses de [Lacroix 95, Dacre-Wright 93, Nashashibi 93, Fillatreau 93]. Les différents modules et l'architecture de contrôle complète seront alors décrits.

1. Le robot ADAM (Advanced Demonstrator for Autonomy and Mobility) mis à la disposition du LAAS est la propriété de Framatome et de Matra Marconi Space.

2. Le projet VAP (Véhicule Automatique Planétaire) a été engagé en 1989 par le CNES, le LAAS y a activement participé à travers le consortium RISP qui regroupe également des laboratoires du CEA, du CNRS, de l'INRIA et de l'ONERA.

3.1 Le robot ADAM

Le robot ADAM (Advanced Demonstrator for Autonomy and Mobility) a été acquis et instrumenté par les sociétés Matra Marconi Space et Framatome dans le cadre du projet Euréka AMR (Advanced Mobile Robot) auquel a contribué le LAAS. Depuis Octobre 1992, il a été confié au LAAS. Ses capacités de locomotion et de calcul embarqué en font un outil bien adapté à l'expérimentation et la validation de navigation en extérieur.

Le châssis, composé de six roues motrices indépendantes non directrices, a été conçu à l'institut VNII Transmash de St. Pétersbourg. Le robot est muni de six capteurs odométriques disposés sur chaque roue et d'une centrale inertielle SFIM. Le calcul de la position, développé par Framatome, et l'asservissement du robot, développé par Matra Marconi Space, sont assurés au moyen de ces capteurs et actionneurs par le module GNC (Guidance Navigation and Control).

Afin de pouvoir appréhender l'environnement, nous avons ajouté à l'équipement du robot une plateforme multisensorielle qui se compose d'un capteur laser à balayage qui permet d'acquérir des images de profondeur 3D, et deux caméras couleur orientables.

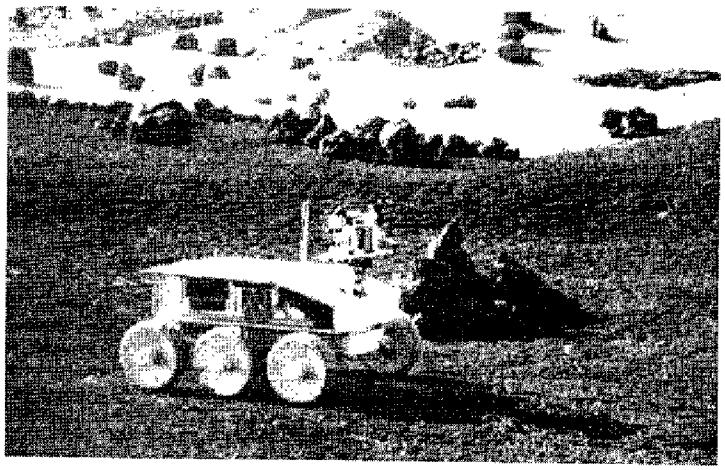


FIG. 3.1 – *Le robot ADAM.*

L'équipement informatique embarqué se compose de deux racks VME équipés de cartes processeurs Motorola MC68030 et MC68040 sous le système d'exploitation VxWorks et de cartes Datacube. Le premier rack intègre les fonctions de contrôle des déplacements et de l'attitude (*i.e.* position et orientation 3D), et le second héberge les fonctions de perception.

3.2 La problématique

La tâche **"Aller à (but)"** qui consiste à rallier de façon autonome un but distant est fondamentale. En effet, quelle que soit la finalité de la mission (cartographie, collecte d'échantillons, extinction d'incendies, installation de matériels, inspection de site, *etc.*), celle-ci comportera nécessairement une ou plusieurs instances de la tâche **"Aller à (but)"**.

Cette tâche de navigation suppose que le robot soit capable de modéliser son environnement, de s'y localiser, de décider des points de perception et des stratégies de déplacement et enfin, de planifier et exécuter les trajectoires. Deux fonctions essentielles qui ressortent sont la *modélisation* de l'environnement et la *planification* des déplacements.

La complexité et la diversité des situations qui peuvent se présenter lors de la navigation en environnement naturel ont conduit à développer une approche adaptative: des stratégies de perception et de déplacement plus ou moins élaborées seront sélectionnées compte-tenu les difficultés présentées par le terrain d'évolution.

La mission consiste à joindre une cible dont un modèle et la position approximative (*e.g.* à 50 mètres au Nord-Ouest) sont initialement transmis au robot. La cible n'est généralement pas dans le champ de vision du robot au début de la mission. Au fur et à mesure de la découverte du terrain, et selon des stratégies de déplacement et de perception choisies selon la nature du terrain, des buts intermédiaires seront déterminés par le robot. L'approche implémentée est une navigation incrémentale réitérée jusqu'à ce que le but soit atteint:

1. Modélisation et classification du terrain selon sa praticabilité.
2. Sélection d'un sous-but dans la zone connue de l'environnement en fonction de la nature des régions à traverser, de façon à s'approcher du but ou à améliorer sa connaissance de l'environnement.
3. Déplacement vers le sous-but selon un mode adapté au terrain.
4. Observation de la cible et mise à jour de sa position.

3.3 Les solutions mises en œuvre

Le projet EDEN devait nous permettre de démontrer par une expérimentation réelle le bien fondé et la viabilité des concepts nécessaires à une navigation autonome en environnement naturel. Les principaux aspects qui ont été mis en valeur sont la mise en œuvre in situ des algorithmes de traitement, la cohabitation de ces traitements dans une architecture logicielle, le contrôle et la coordination de ces traitements par un système décisionnel.

Pour cette première démonstration d'autonomie en milieu naturel, une seule tâche (fondamentale) est accomplie: la tâche **"Aller à (but)"**. L'architecture de contrôle s'en trouve simplifiée, mais se conjugue cependant selon les principes énoncés dans le premier chapitre du mémoire: elle se compose des deux niveaux fonctionnel et décisionnel instanciés dans le cas de l'expérimentation EDEN (figure 3.2 page suivante).

Afin de s'abstraire dans un premier temps de considérations matérielles, et en particulier des capacités de calcul embarqué, une partie des traitements a été déportée sur des stations de travail "au sol".

3.3.1 Les fonctions de base de la couche fonctionnelle

Comme nous l'avons vu, la tâche de déplacement en milieu naturel et a priori inconnu requiert un certain nombre de fonctions fondamentales qui peuvent être instanciées sous la forme de différents algorithmes. Nous allons dans un premier considérer ces différents composants puis nous verrons comment ils ont été intégrés à la couche fonctionnelle d'ADAM.

▷ **Modélisation de l'environnement:** Contrairement aux environnements structurés, et en particulier les environnements d'intérieur qui peuvent être modélisés par des primitives géométriques simples, la complexité des environnements naturels accentue l'importance des choix de modèles de représentation. Ces modèles doivent être adaptés au type de terrain et aux différents processus impliqués dans la navigation: planification de trajectoire 2D ou 3D,

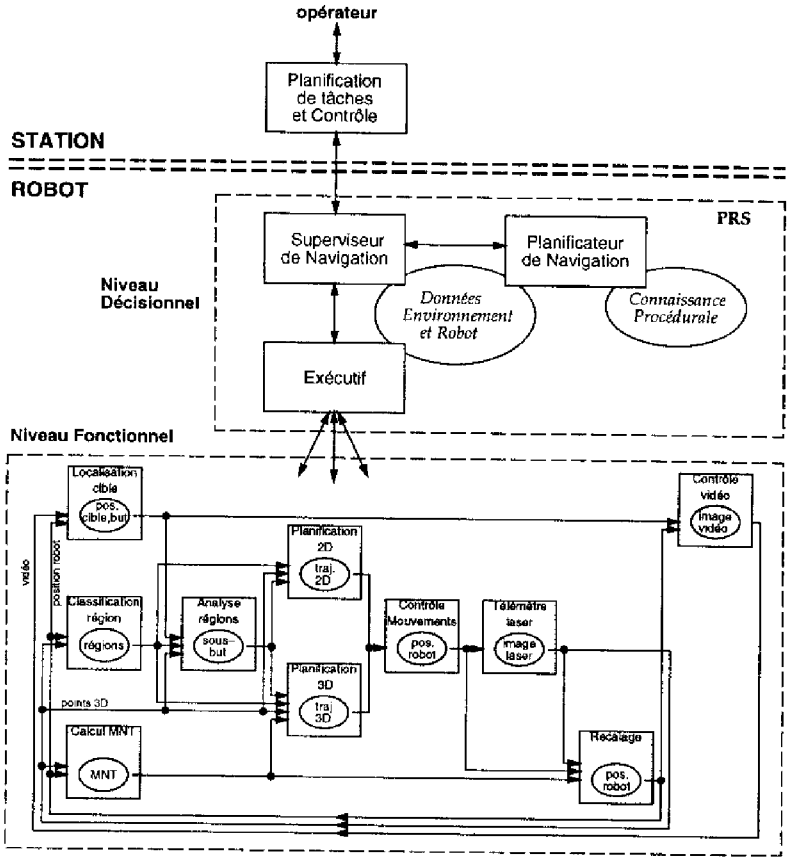


FIG. 3.2 – *L'architecture de contrôle et le flux de données entre les modules.*

localisation sur amers, stratégie de navigation. Ainsi, les images brutes de points 3D obtenues par un balayage laser ou par corrélation en stéréovision [Lasserre 95] vont donner lieu à différentes procédures de modélisation et à différentes représentations qui coexistent dans le système. Un aspect particulier est l'obtention de données locales, au fur et à mesure de la découverte de l'environnement, qui sont fusionnées incrémentalement en une description globale du monde. Les trois fonctions principales qui nécessitent des représentations spécifiques sont:

▷ **La localisation:** Une localisation relative à l'environnement est indispensable afin de corriger les incertitudes cumulées par les méthodes proprioceptives de localisation et permettre la construction incrémentale des modèles de l'environnement dans un repère commun. Dans un environnement naturel, la localisation s'opère relativement à des amers naturels extraits des données perceptuelles et maintenus dans un repère global. Le projet EDEN avait également pour fonction de tester différentes méthodes de recalage: sur des pics extraits d'une représentation d'une carte d'élévation par des B-splines [Fillatreau 93], par corrélation sur la base de cartes d'élévation [Nashashibi 93] ou par appariement d'objets naturels (rochers) [Betge-Brezetz 95] modélisés par des ellipsoïdes ou des super-quadriques [Betge-Brezetz 94].

▷ **La stratégie de navigation:** La stratégie de navigation repose sur une description topologique de l'environnement. La recherche de chemin consiste à explorer un graphe de

régions adjacentes étiquetées selon leur praticabilité (terrain plat, pentu, accidenté, obstacle, inconnu). La classification du terrain en régions s'opère selon une méthode rapide sur la base d'attributs de l'image de points 3D (densité des points, variance sur l'altitude des points, normale moyenne et sa variance) [Lacroix 95]. Une valeur de confiance dans l'étiquetage est attribuée à chaque région qui permet de lever les ambiguïtés lors des procédures de fusion, et de guider la recherche de chemins dans le graphe. La figure 3.3 présente le résultat d'une classification à l'issu de trois perceptions.

FIG. 3.3 – *Classification du terrain après trois perceptions. Les régions sont du plus clair au plus sombre:*

inconnu
plat
pentu
accidenté
obstacle

▷ **La planification de trajectoire:** La planification de trajectoire a elle-même requis l'intégration de trois fonctions de traitement distinctes correspondant à différents modes de navigation sélectionnés selon le type de terrain traversé. En effet, la planification 3D est une opération coûteuse (en temps) qui ne se révèle pas toujours nécessaire:

- En terrain bien dégagé, le mode réflexe permet de se dispenser des opérations de modélisation et de planification. Il consiste à se diriger directement vers le but ou le sous-but en surveillant la présence d'obstacles au moyen d'un balayage laser en avant du robot.
- Lorsque le terrain est plat mais encombré d'obstacles avoisinants, la trajectoire est déterminée à partir du bitmap binaire libre/obstacle. L'algorithme implémenté est celui qui a été présenté dans le cadre du module PLANIF2D de la couche fonctionnelle d'*Hilare2*. Rappelons en deux mots qu'il consiste à déterminer le diagramme de voronoï des obstacles qui permet le passage du robot approximé par un cercle.
- Lorsque la stratégie de déplacement requiert la traversée d'une zone accidentée, dont les irrégularités sont assez marquées pour altérer l'attitude du robot et risquer son déséquilibre ou son blocage, une planification 3D est nécessaire [Siméon 93, Dacre-Wright 93]. Le planificateur utilise une carte d'élévation [Nashashibi 93] (avec une discrétisation de 10cm) et détermine la trajectoire composée de placements successifs qui satisfont les contraintes suivantes: non-collision avec le terrain, stabilité du robot, maintient du contact des six roues avec le sol en considérant le débattement des suspensions, non-holonomie des déplacements entre deux positions. Le planificateur produit incrémentalement un graphe des configurations accessibles par une séquence de commandes élémentaires (translations ou rotations). La trajectoire est recherchée à travers ce graphe

au moyen d'un algorithme A*. Le temps de calcul très dépendant de la complexité du terrain peut prendre plusieurs minutes pour une carte discrète de 150 × 200 sur une station Silicon Graphics Indigo R4000. La figure 3.4 montre une trajectoire calculée lors d'une expérimentation avec le robot ADAM.

FIG. 3.4 – *Une trajectoire 3D planifiée sur une carte d'élévation numérique déterminée à partir d'une fusion d'images de points laser.*

3.3.2 Organisation de la couche fonctionnelle

Les différentes fonctionnalités présentées ci-dessus ont été intégrées dans des modules. La principale difficulté d'un point de vu organisationnel a été de définir les interactions en terme de contrôle et de flux de données. Bien qu'ils n'aient pas été produits par G^{en}oM, ces modules ont été construits en respectant la structure et le formalisme présentés dans les chapitres 1 et 2 de la partie précédente. En particulier, les interactions sont basées sur les protocoles de communication client/serveur et de transfert de données par posters. Ceci nous a permis de produire de façon quasi automatique les fonctions d'interaction entre les modules et le niveau décisionnel constituant les prémices de G^{en}oM. Ainsi, les services sont requis par le jeu des requêtes et des répliques, et les connexions entre les modules sont dynamiquement établies par le niveau décisionnel. Les traitements successivement invoqués lors d'une itération nominale dans le déroulement de la mission, sont les suivants (figure 3.5 page ci-contre):

1. En parallèle:
    1. Recherche de la cible à partir d'images vidéo. Si elle se trouve effectivement dans le champ visuel du robot, son identification permet de mettre à jour sa position et d'affiner la position du but, une position face à la cible, dans le repère global (module Localisation cible).
    2. Classification des régions à partir d'une image de points laser et fusion des régions dans une carte globale (module Classification régions).
3. Recherche d'un sous-but en parcourant le graphe des régions classifiées (module Analyse région).
4. Calcul d'une carte d'élévation numérique du terrain dans le repère cartésien si une région accidentée doit être traversée (module Calcul MNT).
5. Calcul de la trajectoire qui joint le sous-but au moyen du planificateur adéquat (module Planification 2D ou Planification 3D).

6. Exécution de la trajectoire avec asservissement en position calculée par l'odométrie et la centrale inertielle (module Contrôle mouvements, ou GNC). Parallèlement, surveillance d'obstacles par balayage laser (module Télémètre laser).
7. Nouvelles acquisitions laser et vidéo (modules Télémètre laser et Contrôle vidéo).
8. Localisation relative à l'environnement et mise à jour des amers dans une carte globale (module Recalage) .

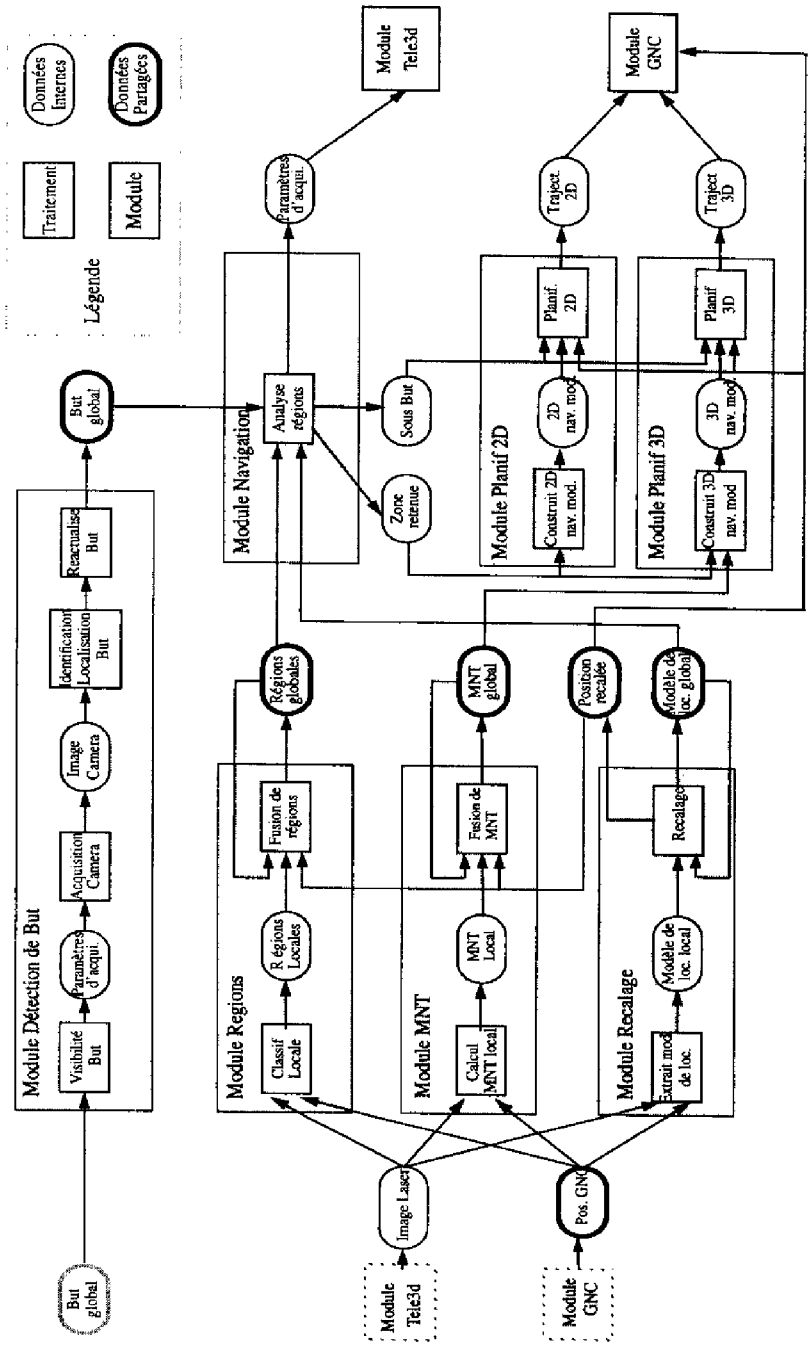


FIG. 3.5 – *Fonctions de traitement et flux de données de la couche fonctionnelle d'ADAM.*

3.3.3 Le niveau décisionnel

La téléprogrammation de robots d'intervention [Chatila 92, Perret 95] et la planification des missions au niveau de la station au sol, qui considère en particulier les contraintes temporelles (durée d'ensoleillement ou les fenêtres de communication avec la station au sol) [Laruelle 94], ont donné lieu à des travaux au LAAS qui n'ont cependant pas encore été intégrés dans le cadre du projet EDEN. On ne les développera donc pas plus en avant.

Le niveau décisionnel est structuré en trois parties. Cependant les missions se réduisant à une tâche unique relativement séquentielle (voir le paragraphe précédent) cette structuration, et en particulier le rôle de coordinateur du superviseur, a été peu mis en valeur au cours de cette expérimentation. Les fonctions principales de chacune de ces entités sont les suivantes:

- **Le superviseur** reçoit de l'opérateur la tâche **"Aller à (cible)"** et la décompose en actions. Cette tâche dépendant du contexte doit être affinée par le planificateur de navigation. Les séquences d'actions déterminées sont transmises à l'exécutif. En fonction des résultats des traitements retournés par l'exécutif le plan nominal d'actions ou des plans alternatifs sont sélectionnés (détection d'un obstacle, ...). Enfin, le superviseur a également la charge de transmettre les comptes rendus d'exécution ou des informations spécifiques (*e.g.* cartes topologiques, images vidéo) à l'opérateur.
- **Le planificateur de navigation.** L'environnement n'étant que grossièrement connu en début de mission, la planification des actions ne peut se faire qu'au fur et à mesure de l'avancée dans la mission et de la découverte de cet environnement. Cette décomposition des tâches en actions exécutables est assurée par le planificateur de navigation. La stratégie de navigation repose essentiellement sur des opérations de recherche de chemins dans le graphe des régions et de calcul de visibilité de la cible qui sont intégrés dans les modules Analyse régions et Localisation Cible. La décision finale reste cependant du ressort du planificateur. Le plan d'actions ainsi produit se compose des différentes opérations d'acquisition, de modélisation, de localisation du robot ou de la cible et de planification de chemins qui ont été précédemment décrites.
- **L'exécutif** démarre les différentes actions auprès des modules de la couche fonctionnelle au moyen des requêtes. Il permet l'exécution en parallèle de plusieurs actions. Les résultats issus des traitements retournés par le biais des répliques finales sont transmis au superviseur. Un traitement non-nominal, marqué par un bilan d'exécution particulier ou le retour de la réplique finale d'une activité de surveillance (*e.g.* détection d'un obstacle pendant un mouvement - voir la procédure **(MOVING)** décrite ci-dessous) a pour effet de terminer la séquence d'actions requise auprès de l'exécutif et d'en référer au superviseur qui décidera alors de la suite des actions.

L'ensemble du niveau décisionnel a été intégré au moyen de C-PRS (chapitre I.1) qui est particulièrement bien adapté à la définition de procédures (ou KA en C-PRS) d'affinement et d'exécution qui peuvent être activées par des invocations spécifiques ou par l'apparition de faits nouveaux dans la base de faits de C-PRS. Les invocations sont des "buts" postés par l'opérateur et par les procédures elles-mêmes. La base de faits, maintenue à jour en permanence par les procédures reflète en particulier l'état des activités de la couche fonctionnelle et l'évolution du système. Elle comporte également les informations nécessaires à la stratégie de navigation produites par les activités.

La figure 3.6 page suivante montre la procédure principale de navigation qui boucle jusqu'à

ce que le but soit atteint. Cette procédure est invoquée par une procédure de plus haut niveau qui procède à l'initialisation de la mission (enregistrement du modèle et de la position estimée de la cible). On remarque que les opérations de mise à jour du graphe des régions et de la position de la cible y sont effectuées en parallèle.

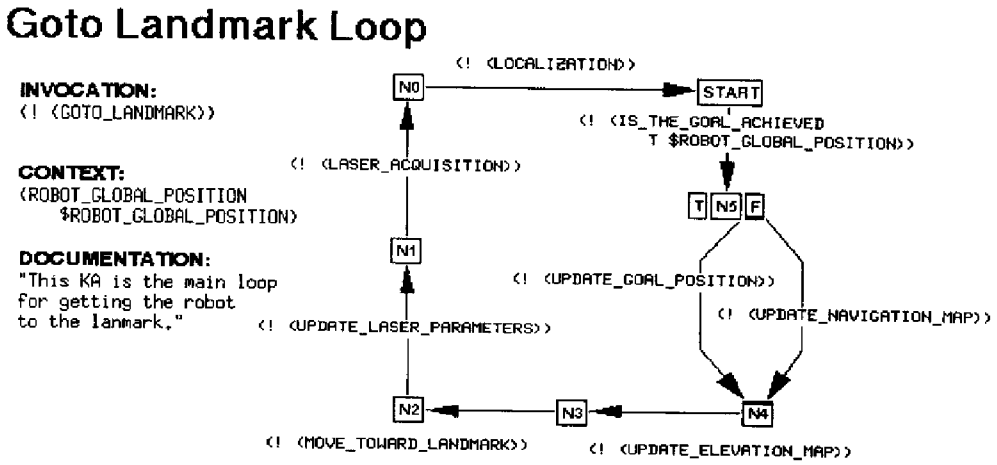


FIG. 3.6 – *La boucle principale de navigation.*

L'exécutif intègre les procédures d'interaction avec les modules de la couche fonctionnelle. Pour cela une bibliothèque de procédures C-PRS a été générée au moyen de la version préliminaire de G^{en}oM. Ces procédures, également dénommées KA-actions, établissent l'interface entre C-PRS et les fonctions d'émission et de réception des requêtes et des répliques associées à chaque module.

Ainsi, la procédure de la figure 3.7 page suivante, qui permet à l'exécutif d'exécuter une trajectoire 2D tout en surveillant la présence d'obstacles, déclenche les actions correspondantes par l'émission de requêtes auprès des modules Télémètre laser et Contrôle mouvements. L'exécution de la trajectoire n'est lancée qu'après confirmation, par la réplique intermédiaire, du démarrage de l'action de surveillance (nœud N3). Ces deux traitements s'exécutent alors en parallèle jusqu'à la terminaison de l'une ou l'autre action, marquée par la réception de la réplique finale qui aura pour effet d'interrompre l'action encore en cours. Ainsi, la fin de l'exécution de la trajectoire terminera la procédure de surveillance, et réciproquement, la détection d'un obstacle entraînera l'arrêt du robot.

3.4 Conclusion

Cette intégration s'est traduite par plusieurs expérimentations sur différents sites. Une cible disposée à 30m de la position initiale dans un environnement encombré est rejointe en 20mn dans le mode de planification 2D. Ce délai est essentiellement dû à la vitesse de déplacement limitée du robot ADAM (28 cm/s) et au temps d'acquisition des images de points 3D par le laser.

Cette expérience a permis de mettre en œuvre la couche fonctionnelle dans un contexte difficile pour la robotique: environnement inconnu, données sensorielles riches mais volumineuses, fonctions de traitement complexes.

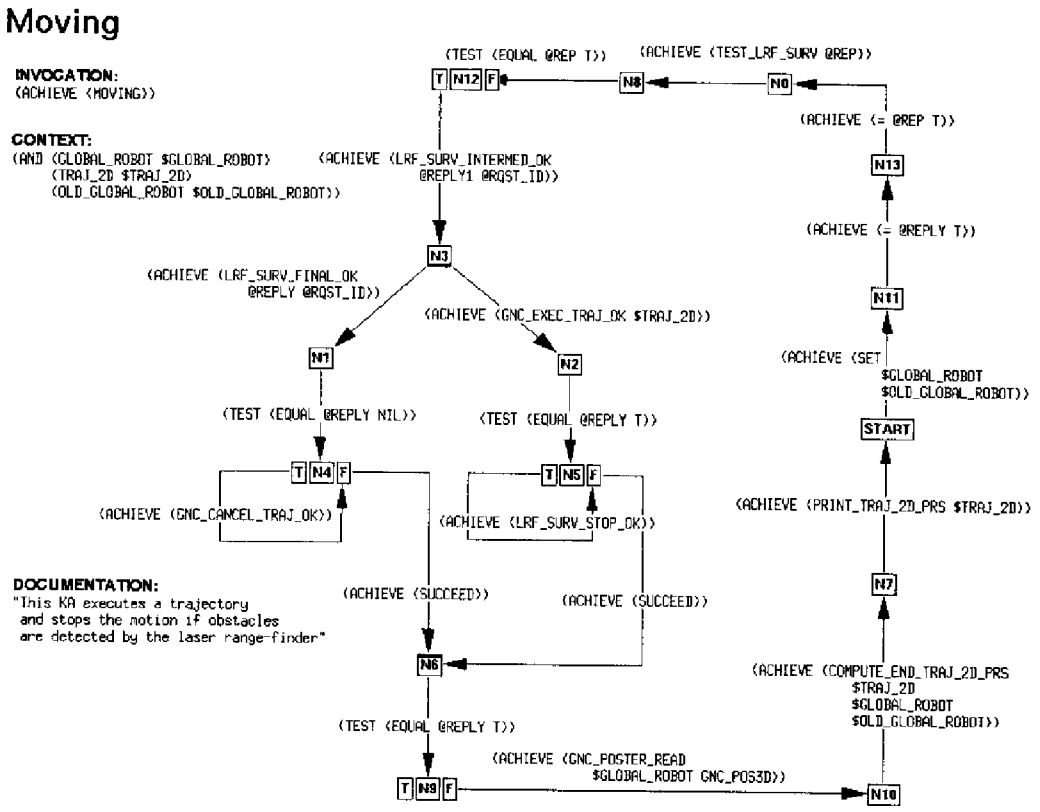


FIG. 3.7 – *La procédure d'exécution de trajectoire 2D avec surveillance de présence d'obstacle.*

Si la mission se résume pour le moment à une tâche (**"Aller à (but)"**), elle permet cependant d'expérimenter de façon étendue la couche fonctionnelle: échange des données, parallélisme d'exécution des fonctions, interactions avec l'exécutif, actions réflexes programmées au niveau de l'exécutif, *etc.* En plus des progrès algorithmiques qu'elle a permis dans les domaines de la perception et de la modélisation d'un environnement naturel, de la planification de trajectoires 3D, du choix de stratégies de perception et de déplacements et de la localisation, cette expérience a été une étape importante dans la mise en œuvre de l'architecture de contrôle proposée au LAAS.

Les procédures d'interaction entre C-PRS et les modules ont été partiellement générées au moyen d'une première version de G^{en}oM. On verra cependant une implémentation plus systématique dans le cadre du projet multi-robots STRADA présenté dans le chapitre suivant. Elle permettra des interactions plus dynamiques entre les niveaux décisionnel et fonctionnel. En effet, comme le montre la figure 3.7 les communications intégrées dans les procédures de l'exécutif sont bloquantes (attente des répliques). Suite à cette expérience nous avons mis en place au niveau de l'exécutif une méthode générale de réception des événements et données issus des activités des modules qui sont directement enregistrés dans la base de faits de C-PRS.

Dans le cadre de la robotique en environnement naturel, de nouveaux développements sont en cours d'élaboration en collaboration avec Alcatel Espace et qui doivent se traduire par des expérimentations sur le robot LAMA.

Chapitre 4

STRADA: Une application multi-robots

La coopération d'un grand nombre de robots est un défi peu fréquemment relevé et qui pourtant, outre un côté incontestablement grisant, ouvre des débouchés applicatifs nouveaux et va nous permettre de confronter à la réalité la théorie de la coopération multi-agents et l'autonomie des robots en environnement dynamique partiellement structuré. Il va sans dire que la robustesse et la cohérence des robots sont dans ce contexte primordiales.

Dans ce cadre, nous avons collaboré à un projet européen ESPRIT III: le projet MARTHA (Mobile Autonomous Robots for Transportation and Handling Application), et mis en place un projet interne d'expérimentation complète baptisé STRADA.

Le projet MARTHA (EP 6668) est une collaboration européennne[1] qui a débuté en Juin 1992 et devrait se conclure par une première intégration sur deux démonstrateurs in situ, au centre d'expérimentation de la SNCF à Trappes sur robot COMMUTOR (photo 4.1 page suivante) et à l'aéroport de Francfort sur robot INDUMAT, à la fin 1995. Son objet est l'étude et la conception d'un système automatisé de transitique: une flottille de robots doit procéder au transfert des containers dans des ports, des aéroports ou des gares de triage. Soulignons qu'un système semi-automatisé intégrant 50 véhicules est déjà en service au nouveau port de Rotterdam (photo 4.2 page suivante). Dans le cadre de ce projet nous sommes plus spécifiquement responsables du planificateur de chemins, de la coordination multi-robots et du superviseur de chaque robot.

Le projet interne STRADA vise à une intégration complète d'une coopération multi-robots au laboratoire. La flottille sera bien évidemment réduite! Un troisième robot de la famille Hilare a été instrumenté à cette fin (multi ≥ 3). Cependant, les principes généraux ont été strictement respectés et, afin de procéder à des tests plus intensifs concernant la coopération, une simulation réaliste et complète a également été développée.

Ce projet d'expérimentation multi-robots est avant tout un formidable travail d'équipe, emmené par Rachid Alami, dont je vais présenter les grandes lignes (on pourra aussi se référer aux publications suivantes: [Aguilar 95, Alami 94, Alami 95a]). Ma contribution concerne plus particulièrement l'organisation et l'intégration des modules de la couche fonctionnelle et ses intéractions avec le superviseur de robot, tant pour la simulation d'une flotte de robots que pour l'expérimentation à trois robots. Je pense pouvoir affirmer que la réalisation de

1. Partenaires: **en France:** FRAMATOME, SNCF, ROL, PROMIP (LAAS, Midi-Robots); **en Allemagne:** MANNESMAN, INDUMAT, Aéroport de Francfort, Université de Karlsruhe; **aux Pays bas:** Port de Rotterdam; **en Espagne:** IKERLAN.

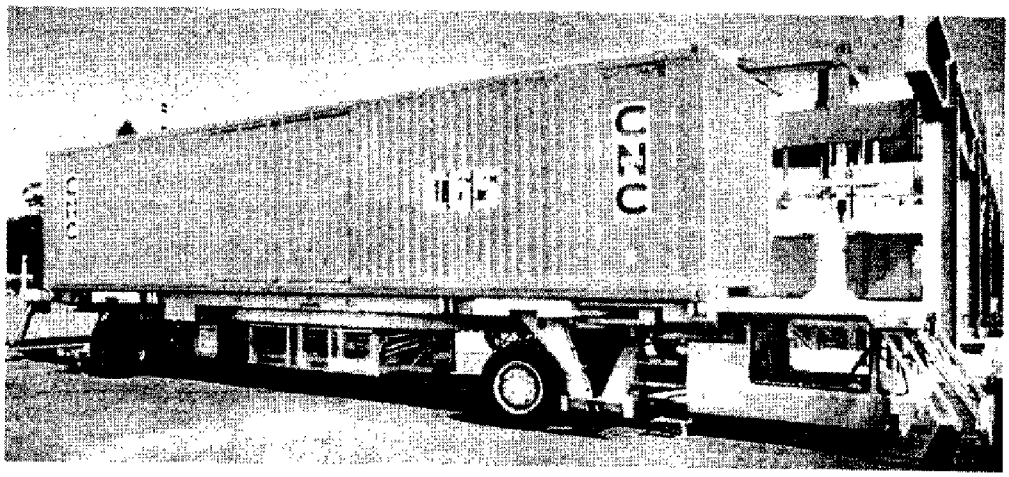


FIG. 4.1 – *Le robot Commutor.*

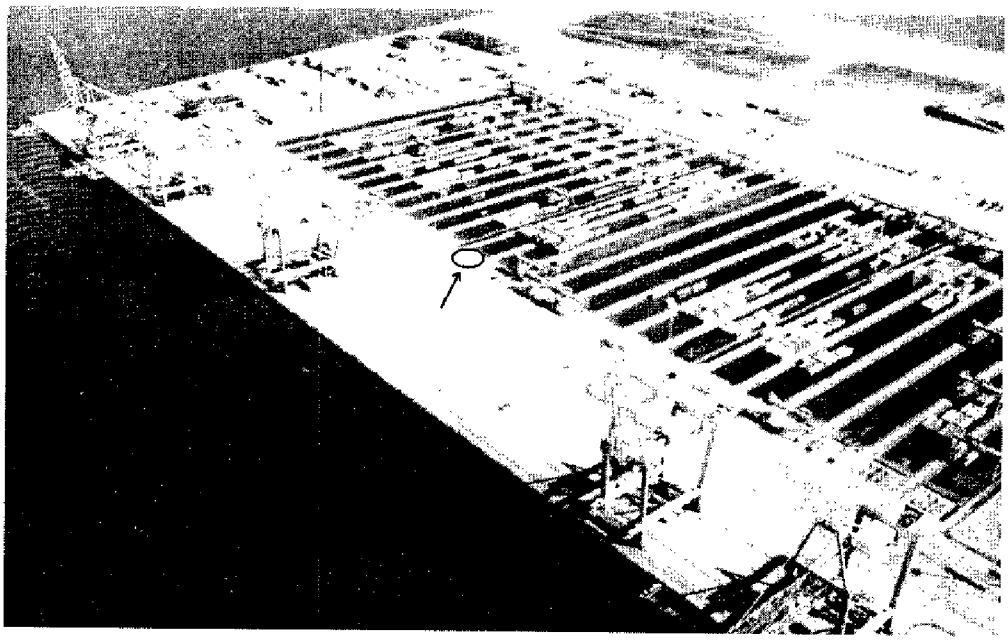


FIG. 4.2 – *Le nouveau port de Rotterdam. Le cercle indiqué par la flèche montre un véhicule de dimensions similaires à Commutor.*

ce double projet eut été très difficile sans le générateur de modules. En effet, comme nous allons le voir, de nombreux modules faisant intervenir des algorithmes complexes utilisés et/ou conçus par une dizaine de personnes, ont été nécessaires. Outre la génération automatique des modules, le formalisme et le cadre conceptuel offert par $G^{en}{}_{o}M$ aident à une coordination des travaux, en particulier grâce à la spécification précise des fonctionnalités et de leurs interfaces. La description formelle des modules a découlé naturellement de la définition, certes ardue, du cahier des charges du projet rendant ainsi les serveurs disponibles pour une première intégration avant même la mise au point des algorithmes.

4.1 La problématique générale

Le nœud du projet est la conception d'un système qui permet de faire évoluer un grand nombre de robots dans un réseau routier où peuvent survenir un certain nombre d'imprévus tels que la présence de personnes, d'autres véhicules, de robots en panne. Les aspects critiques du système qui sont:

- La planification des tâches de transitique.
- La coordination entre les véhicules.
- La planification de chemins.
- L'exécution des trajectoires avec détection d'obstacles et une localisation précise (incertitudes bornées).

La coopération multi-robots a donné lieu à de nombreux travaux théoriques dont les approches très diverses attribuent des sens différents au terme de "coopération". Ces travaux vont de la généralisation de la planification de chemins à N mobiles (Espaces de Configurations Généralisés), à la problématique générale de la coopération multi-agents, en passant par la (re)définition de codes de la route. La majorité de ces contributions propose des systèmes centralisés dont la complexité impose de ne considérer qu'un nombre limité de situations ou un nombre restreint de robots.

Bien que cela s'écarte du travail présenté dans ce mémoire il nous paraît important, avant d'aborder l'architecture de contrôle et la couche fonctionnelle, de pouvoir apprécier la difficulté du problème et les solutions, générales mais concrètes, élaborées par l'équipe qui ont pu être harmonieusement intégrées dans notre architecture. Ces points sont l'objet de cette section.

4.1.1 Vers un système décentralisé

Le nombre important de véhicules et le non-déterminisme de l'ensemble des situations qui peuvent survenir ont conduit à définir un protocole de coordination générique qui permet de résoudre les conflits *localement* au fur et à mesure de leurs occurences, sans remettre en cause le plan global et sans interrompre l'exécution des autres robots.

En effet, une centralisation de l'ensemble des synchronisations se traduirait par une suractivité du système central de coordination et une portée limitée des plans de chaque robot ou des contraintes à long terme qui ralentiraient l'ensemble du flot. C'est pourquoi nous préconisons un système décentralisé qui permet de résoudre les conflits localement par les robots mis en cause.

Le système se compose d'un ensemble de robots autonomes et d'une station centrale dont le rôle se restreint à l'attribution de missions et de routes pour répartir le trafic, sans spécifier ni le chemin géométrique ni l'ordre de passage des robots: *"Prendre le container C_i de la station S_j et le déposer à la station S_k en passant par la voie V_l"*. La coordination entre les robots est exclusivement à la charge des robots.

Les robots doivent donc disposer des capacités de planification, de localisation, d'évitement d'obstacle, ... propres aux robots autonomes évoluant seuls (voir le premier chapitre de cette partie), auxquelles doivent s'ajouter des capacités spécifiques à la coopération de robots.

4.1.2 Le modèle de l'environnement

Afin de pouvoir raisonner sur les ressources spatiales que vont se partager les robots, la représentation de l'environnement est structurée en différentes entités topologiques. Le modèle topologique est constitué d'*aires*, de *voies* et de *carrefours*. Les aires d'évolution hébergent les *stations* de chargement, de déchargement et de stationnement. Les voies, à sens unique, connectent les aires et s'intersectent aux *carrefours*. Pour rendre plus efficace la gestion des ressources, les voies et les carrefours sont eux-mêmes structurés en entités plus petites: les *cellules* qui sont les éléments de base dans la résolution des conflits locaux. Globalement, les robots naviguent à travers un graphe orienté de cellules qui, en mode nominal, contiennent au plus un robot. Une aire est une cellule unique dans laquelle, à l'exclusion des stations, plusieurs robots peuvent évoluer simultanément. La présence de plusieurs robots en une même cellule va requérir une coopération plus fine. La figure 4.3 montre un environnement structuré et le modèle topologique qui lui est associé.

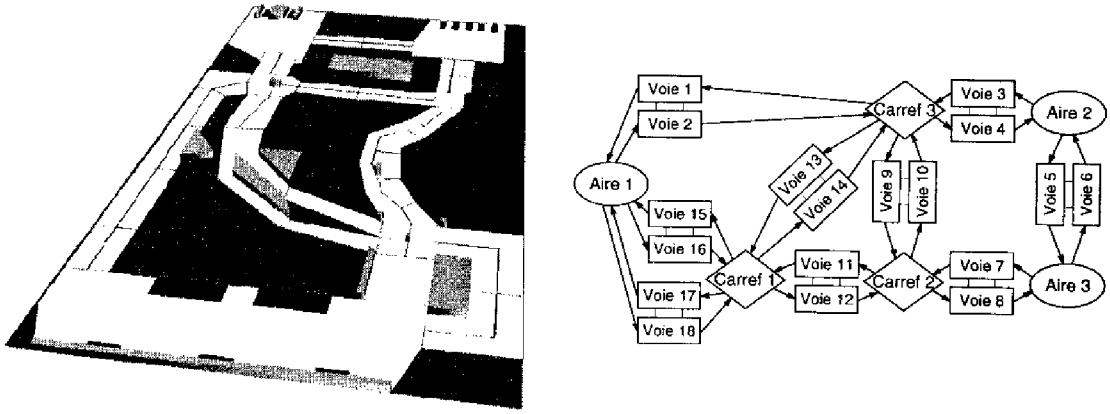


FIG. 4.3 – *Un environnement et son modèle topologique.*

Le modèle de l'environnement fourni à chaque robot est une description hiérarchique qui contient des informations topologiques et géométriques:

1. **Un réseau de routes qui connecte les aires.** Ce niveau de description est l'unique information dont dispose la station centrale pour élaborer les routes des robots.
2. **Un graphe orienté de cellules** dont l'orientation définit les sens privilégiés (mais non-exclusifs) des mouvements le long des voies et aux carrefours.
3. **Une description géométrique** des cellules (des zones polygonales).
4. **Une description des amers** (pour le recalage), **des obstacles fixes** et **des stations.**

4.1.3 Un paradigme d'insertion de plans

La coopération est basée sur un paradigme général selon lequel les robots fusionnent leur plans incrémentalement à un ensemble de plans déjà coordonnés grâce à des échanges d'informations concernant l'état courant et les actions futures: le Paradigme d'Insertion de Plans (PIP). On trouvera de plus amples informations dans [Alami 95c, Alami 95d, Robert 96].

Dans le contexte de la coordination multi-robots, une instance particulière du PIP, relative à l'allocation de ressources spatiales que sont les cellules, a été développée. Schématiquement, un robot annonce les cellules dont il a besoin et reçoit en retour un ensemble de plans

coordonnés des robots qui ont déjà planifié l'utilisation des cellules mentionnées. Il procède alors à une Opération d'Insertion de Plan (OIP) qui consiste à insérer ses actions parmi celles des plans reçus, ces plans ayant déjà été validés par des OIP faites précédemment par d'autres robots. Dans ce contexte, une action correspond à l'allocation ou à la libération de ressources. Les actions sont insérées de façon à ne pas contraindre les actions des plans réceptionnées qui peuvent être déjà en cours d'exécution. Le plan global ainsi produit est un graphe orienté dont les nœuds marquent les début et fin d'action, et les arcs les relations de précédence. Un cycle dans le graphe traduit un interblocage qui est ainsi immédiatement détecté.

Des OIP peuvent ainsi s'exécuter simultanément à la condition qu'elles concernent des ensembles disjoints de ressources. Cet aspect est primordial car cela permet de résoudre *localement* la majorité des conflits. En effet, dans le contexte d'un réseau routier, l'occurence de plusieurs "petits" conflits, *i.e.* impliquant un nombre restreint de véhicules, en des lieux distants, est la situation la plus fréquente.

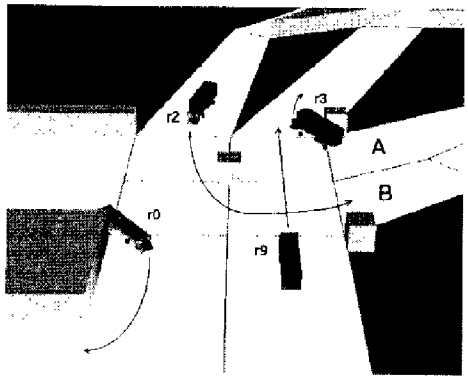
Lorsqu'une coordination ne peut être réalisée sur la base des cellules, la coordination s'effectue à un niveau plus fin: celui des trajectoires, ce qui définit une hiérarchie d'OIPs:

1. **OIP pour l'occupation de cellules:** dans la mesure du possible une cellule est attribuée à au plus un robot. Ceux-ci peuvent ainsi planifier leurs trajectoires indépendamment les uns des autres à l'intérieur des cellules attribuées. L'OIP s'opère alors au niveau topologique (cellules). Afin de ne pas contraindre inutilement les autres robots, la stratégie d'allocation ne réserve qu'une cellule à l'avance sur les voies et l'ensemble des cellules nécessaires pour traverser et quitter un carrefour (qu'il faut éviter de bloquer). La figure 4.4 page suivante présente un exemple de synchronisation à un carrefour.
2. **OIP au niveau trajectoire:** dans les aires d'évolution ou lors de la présence d'obstacles imprévus sur une voie ou un carrefour, les robots sont amenés à manœuvrer dans des cellules communes. Dans ce cas une coopération plus fine, basée sur le même protocole mais sur un autre planificateur, est appliquée pour coordonner les actions au niveau des *trajectoires*. La figure 4.5 page suivante présente un exemple de synchronisation sur trajectoire.

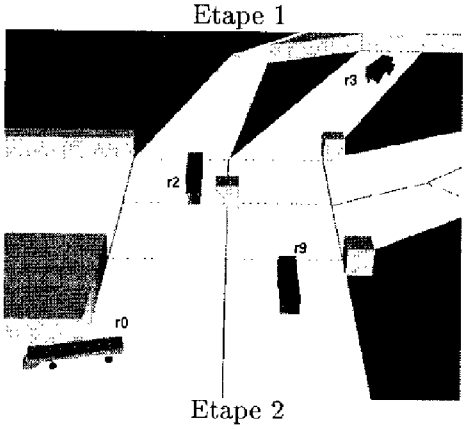
Il est important de souligner que les synchronisations ne sont pas indexées sur des instants, mais sur le passage effectif du robot par certaines positions: lors d'un changement de lieu topologique ou lors du franchissement d'une abscisse curviligne donnée d'une trajectoire. En effet, des hypothèses quant à la durée d'une action seraient trop aléatoires: il est difficile d'estimer les temps de parcours qui varient en fonction des obstacles rencontrés et du nombre de coordinations, ou encore le temps nécessaire au chargement et déchargement des containers, qui dépend notamment de la disponibilité de la grue et des aléas de la manutention.

4.1.4 L'architecture de contrôle d'un robot

L'architecture de contrôle se décline selon l'architecture générale d'un robot mobile autonome présentée dans le premier chapitre du mémoire: elle comporte un niveau décisionnel et un niveau fonctionnel. Le niveau décisionnel ou le *superviseur du robot* est structuré en niveaux hiérarchiques, depuis la planification de mission jusqu'à l'exécution des activités, et intègre des fonctions spécifiques à la coordination multi-robots. Ces trois niveaux, qui s'exécutent en parallèle, sont selon un ordre hiérarchique décroissant: *le niveau mission*, *le niveau de coordination* et *le niveau d'exécution* (voir la figure 4.6 page 133).

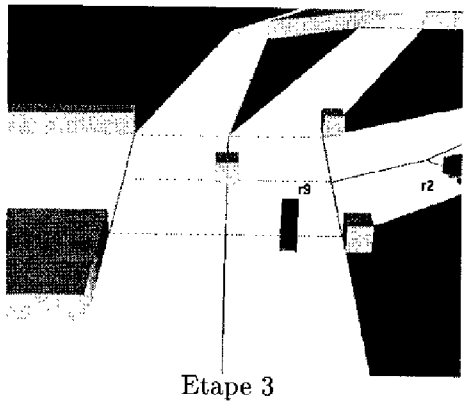


Etape 1



Etape 2

FIG. 4.4 – **Coordination à un carrefour:** *Les robots r3 et r0 ayant des listes de ressources disjonctives vont pouvoir procéder à l'OIP en parallèle et traverser le carrefour sans se synchroniser* (étape 1). *Le robot r9 doit attendre que r3 ait planifié de quitter la cellule au delà du carrefour. r2 passe donc avant r3, après une synchronisation avec r0* (étape 2). *Enfin, r2 libère le carrefour que peut alors traverser r9* (étape 3).



Etape 3

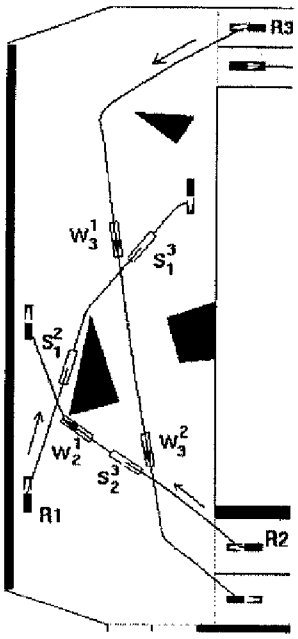


FIG. 4.5 – **Synchronisation sur trajectoires.** *Les robots R_1, R_2 puis R_3 procèdent dans cet ordre à un PIP et une planification de trajectoire. La position sur laquelle R_i doit s'arrêter et attendre un événement de R_j est notée W_i^j ("i Waits j"). R_j devra émettre cet événement au passage de la position S_j^i ("j Signals i"). Si le robot j produit l'événement S_j^i avant que le robot i n'est atteint la position d'attente W_i^j, alors le robot i pourra continuer son chemin sans marquer d'arrêt.*

Nous présentons ici rapidement ces trois nivaux. On trouvera un exemple de mission et ses différents stades d'affinement dans [Alami 95a]. Les termes d'une mission précisent les

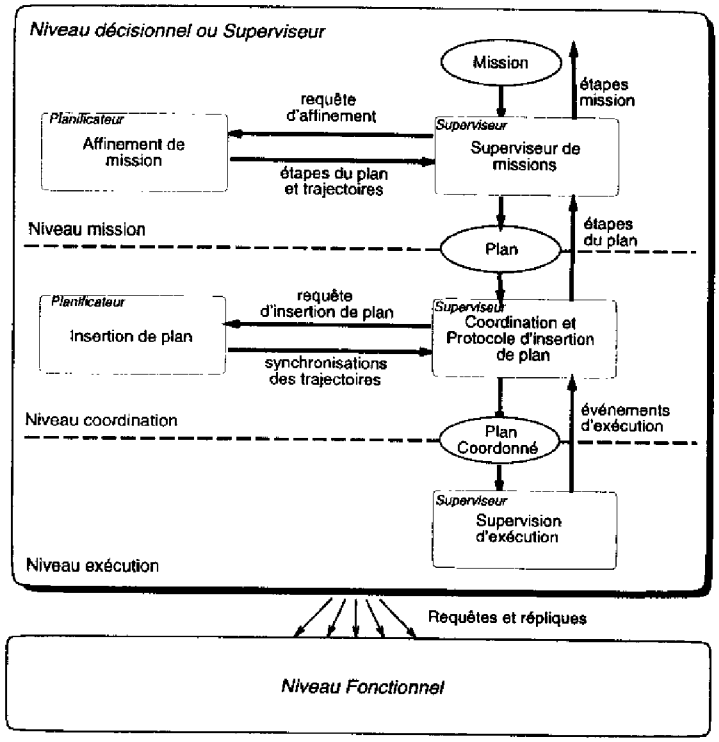


FIG. 4.6 – *Architecture du superviseur du robot.*

stations qu'il faut successivement rallier, les opérations à y réaliser (prise ou dépose de tel container), et éventuellement les routes (voies) privilégiées à emprunter (régulation globale du trafic[2]).

1. **Le niveau mission** est en interaction avec la station centrale. Il interprète et affine les missions sans considérer, dans un premier temps, les conflits avec les autres robots. Le *planificateur de mission* produit un plan d'actions sous la forme d'une succession de trajectoires et d'événements d'entrée/sortie de cellules qui permettront de suivre l'exécution et de se synchroniser avec les autres robots. Le *superviseur de mission* gère le planificateur de mission et, *parallèlement*, contrôle le déroulement des actions déjà planifiées. En cas d'échec (*e.g.* obstacle incontournable en restant dans les ressources attribuées, ...), un plan alternatif est affiné et tenté.
2. **Le niveau de coordination** produit et contrôle le déroulement de plans coordonnés et gère les interactions avec les autres robots. Les plans issus du niveau mission sont incrémentalement validés dans le contexte multi-robots en appliquant le protocole de fusion de plans. L'OIP est exécuté par le *planificateur* qui produit un plan coordonné d'actions synchronisées par les événements à transmettre ou à attendre des autres robots. Le *superviseur* gère les intéraction avec les autres robots et contrôle le déroulement des actions transmises au niveau d'exécution. Un échec d'exécution rapporté par un autre robot dont on attend un événement de synchronisation a pour effet de requérir une nouvelle OIP au près du planificateur. Là encore, les opérations de planification (OIP) s'effectuent parallèlement à l'exécution du plan précédent.
3. **Le niveau d'exécution** exécute les actions coordonnées. Il est *responsable des in-*

2. Chaque robot dispose cependant des fonctions nécessaires pour trouver une route.

teractions avec le niveau fonctionnel. Il doit déclencher et suivre le déroulement des actions et des surveillances mais également surveiller et réagir à un certain nombre d'événements critiques tels que la détection d'obstacles imprévus, l'état du système, les compte-rendus d'échec retournés dans les répliques et les événements de synchronisation ou d'échec transmis par les autres robots.

L'ensemble du superviseur a été développé par Frédéric Robert et codé à l'aide du système de raisonnement procédural C-PRS ([Ingrand 92a]). Chacun des éléments qui composent ces trois niveaux ont été intégrés sous la forme de processus indépendants qui s'exécutent en *parallèle.* Nous examinerons plus loin comment interagissent le superviseur et les éléments de la couche fonctionnelle.

4.2 La couche fonctionnelle

Les fonctions requises à bord des robots ont été complètement spécifiées dans le cadre du projet MARTHA, indépendamment de l'existant, en se focalisant sur le problème spécifique de la coopération. Cependant, nombre des services ayant trait à l'autonomie du robot seul ont déjà été intégrés sous des formes un peu différentes sur nos robots (voir le premier chapitre de cette partie). Après une présentation détaillée de ces nouveaux modules, nous verrons comment, dans le cadre du projet STRADA, ils ont été intégrés en tant que modules clients des modules de base des robots *Hilare* (qui ont fait leurs preuves).

Huit modules ont été définis ([Fleury 95a, Alami 95b]). La fonction principale de chacun d'entre eux est indiquée dans le tableau ci-dessous:

ME	Motion Execution	Asservissement du véhicule
SMC	Sensor based Motion Control	Evitement d'obstacles
PER	Perception	Localisation et modélisation d'obstacles
MP	Motion Planner	Planification coordonnée de chemins
MON	Monitoring	Surveillance des ressources occupées
CSC	Central Station Communication	Communication avec la station centrale
IRC	Inter Robot Communication	Communication avec les autres robots
ADS	Application Dependent Services	Chargement/déchargement, stationnement

Nous allons présenter ces différents modules en nous focalisant sur la façon dont les traitements ont été intégrés, ce qui va permettre d'illustrer la grande variété de comportement des activités (exécutions parallèles, séquentielles, conditionnelles). Les requêtes offertes par chaque module seront listées dans des tableaux récapitulatifs. Les requêtes de contrôle sont notées **C**, celles d'exécution **E** et lorsqu'il peut y en avoir plusieurs occurences **nE** en parallèle.

L'ensemble des modules MARTHA, leur relation client/serveur et les flux de données sont résumés par la figure 4.7 page ci-contre.

4.2.1 Le module ME

La fonction principale du module ME est l'asservissement du véhicule sur des trajectoires transmises par la requête d'exécution `ME_ADD_PATH`. Dans l'optique de planification à court terme au fur et à mesure de l'attribution de ressources spatiales, cette requête a pour effet de

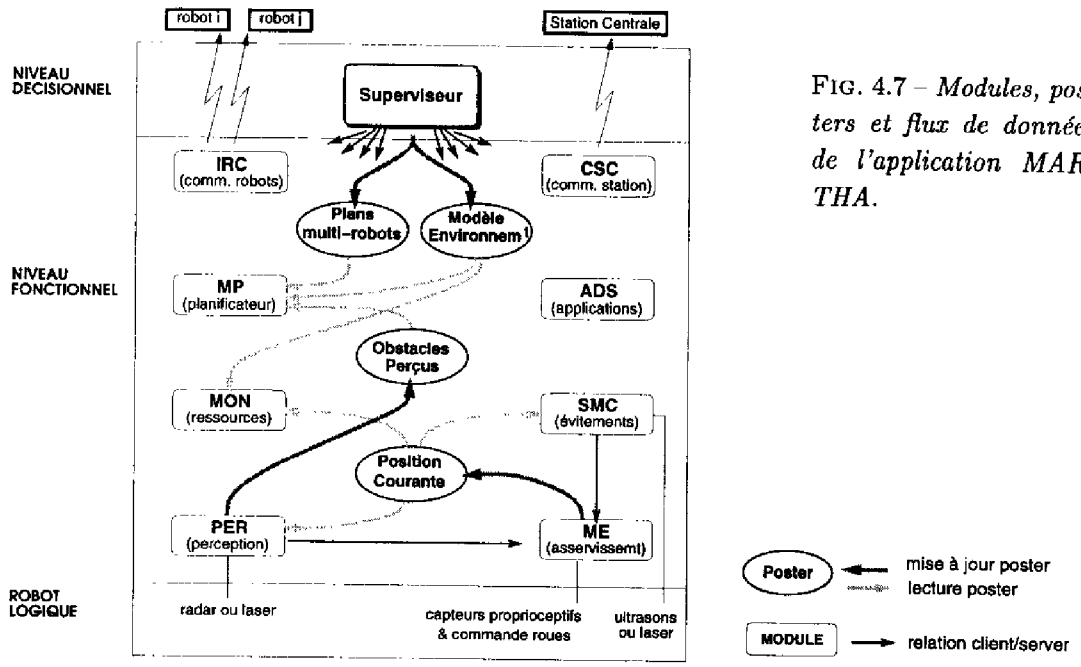


FIG. 4.7 – *Modules, posters et flux de données de l'application MARTHA.*

démarrer l'exécution de la trajectoire si le robot est à l'arrêt, ou de concaténer la nouvelle trajectoire à la précédente dans le cas contraire, si possible sans marquer d'arrêt. Ainsi, plusieurs instances de la requête `ME_ADD_PATH` peuvent être présentes simultanément, mais seule l'une d'entre elle sera dans la phase principale d'exécution **exec**. Les activités en instances sont maintenues dans l'état **start** jusqu'à ce que la portion de trajectoire qui précède se termine. L'interruption (par requête explicite `ME_STOP`) ou l'echec (*e.g.* choc arceau, asservissement défaillant) d'une telle activité à pour conséquence la terminaison des activités en attente. Ces activités d'exécution séquentialisée nécessitent, après une terminaison anormale, un protocole de re-synchronisation entre le client et le serveur qui est assuré par la requête `ME_SYNCHRO`.

Les synchronisations sur trajectoire des robots requièrent l'émission d'événements sur des positions de la trajectoire marquées par leur abscisse curviligne. Le passage de ces positions est détecté par des activités de surveillance activées par la requête `ME_MON_LENGTH` qui prend en paramètre un identificateur de trajectoire (attribué par la requête `ME_ADD_PATH`) et la longueur surveillée. Cette requête peut avoir plusieurs instances et l'activité associée se termine soit sur le passage de la position surveillée, soit avec la terminaison normale ou anormale de la trajectoire.

Ce module maintient également la position du robot dans le poster `POSITION`. Notons qu'en cas de panne le robot continue à participer au PIP et sa position est prise en compte lors de la planification par les autres robots.

ME_INIT	(E)	Initialisation
ME_ADD_PATH	(nE)	Exécution/concaténation de trajectoire
ME_STOP	(C)	Interruption d'exécution de mouvement
ME_SET_VELOCITY	(C)	Changement de la vitesse maximale
ME_MON_LENGTH	(nE)	Surveillance de passage d'abscisse curviligne
ME_KILL_MON	(C)	Interruption d'une surveillance
ME_SYNCHRO	(C)	Resynchronisation après échec
ME_SET_POS	(C)	Recalage

4.2.2 Le module SMC

Le module SMC est, par son interface, très similaire au module ME. Ce qui le distingue est sa capacité de détecter les obstacles imprévus et de les contourner lorsque cela est possible. En effet, pour ne pas gêner les autres véhicules, le robot devra lors du contournement se restreindre aux ressources qui lui ont été allouées et dont les limites sont spécifiées par la requête SMC_SET_BOUNDARY. Lorsque plusieurs robots se partagent une cellule, cette limite est définie par un écart maximal à la trajectoire. Si ce contournement ne peut être exécuté dans les limites fixées, alors l'exécution est interrompue (bilan: PATH_BLOCKED). Le superviseur devra replanifier une trajectoire après modélisation de l'obstacle. L'exécution pourra reprendre après une resynchronisation entre le module et le superviseur (requête SMC_SYNCHRO). On constate que nous disposons de trois niveaux de reprise en cas d'obstacles:

1. évitement local sur la base de données proximétriques
2. évitement après modélisation de l'obstacle
3. nouvelle planification (attribution d'autres ressources spatiales)

Les surveillances de passage d'abscisse curviligne SMC_MON_LENGTH ne peuvent se déclencher que lorsque le robot circule sur le chemin initialement planifié, qui peut n'être rejoint que sur la position terminale.

En mode mission, le superviseur du robot ne s'adresse qu'à SMC qui est lui-même client du module d'asservissement ME.

SMC_INIT	(E)	Initialisation
SMC_ADD_PATH	(nE)	Exécution/concaténation de trajectoire
SMC_SET_BOUNDARY	(C)	Limites d'évolution du robot
SMC_STOP	(C)	Interruption de l'exécution
SMC_SYNCHRO	(C)	Resynchronisation après échec
SMC_MON_LENGTH	(E)	Surveillance de passage d'abscisse curviligne
SMC_KILL_MON	(C)	Interruption d'une surveillance
SMC_GUARDED_MOTION	(C)	Mode de mouvements contraints
SMC_SET_VELOCITY	(C)	Changement de la vitesse maximale

4.2.3 Le module PER

Le module PER a la double fonction de recaler le robot sur des amers prédéfinis (PER_LOCALIZE) et de modéliser des obstacles détectés par le module SMC (PER_GET_NEW_OBSTACLE). Il permet en outre de détecter la présence d'obstacles mobiles dans la zone d'évolution du robot (PER_MON_MOVING_OBJECTS), ce qui aura pour effet d'arrêter le véhicule. Celui-ci ne

pourra repartir que lorsqu'il n'y aura plus de mouvement dans les environs du robot (PER_MON_NO_MOVING_OBJECTS).

PER_INIT	(E)	Initialisation
PER_SET_POS	(C)	Recalage
PER_LOAD_LANDMARKS	(C)	Adjonction d'amers
PER_LOAD_OBSTACLES	(C)	Adjonction d'obstacles
PER_LOCALIZE	(E)	Localisation sur amers
PER_GET_NEW_OBSTACLE	(E)	Modélisation des obstacles avoisinants
PER_MON_MOVING_OBJECTS	(nE)	Surveillance des obstacles mobiles
PER_MON_NO_MOVING_OBJECTS	(nE)	Surveillance de l'immobilité des obstacles
PER_KILL_MON	(C)	Interruption d'une surveillance
PER_SYNCHRO	(C)	Resynchronisation après échec

4.2.4 Le module ADS

Le module ADS intègre les fonctions spécifiques du robot et des stations de transbordement tels que la prise et la pose de container (par grues) ou les opérations de mise en position et de sortie du véhicule des stations.

ADS_INIT	(E)	Initialisation
ADS_DOCK	(E)	Manœuvre de stationnement dans une station
ADS_UNDOCK	(E)	Manoeuvre de sortie d'une station
ADS_PICKUP	(E)	Prise de container
ADS_PUTDOWN	(E)	Dépose de container
ADS_STOP	(C)	Interruption d'une exécution
ADS_GET_ENERGY_LEVEL	(C)	Niveau des batteries ou d'essence
ADS_SYNCHRO	(C)	Resynchronisation après échec

4.2.5 Le module MP

Le module MP est un planificateur de chemins qui intègre un planificateur topologique, un planificateur géométrique et un séquenceur multi-robots. Il permet non seulement de déterminer des chemins topologiques et géométriques, mais également de marquer ces chemins par des points de synchronisation qui correspondent soit à l'attente d'un événement d'un autre robot (attente d'une ressource), soit à l'émission d'un événement vers un autre robot (libération d'une ressource).[3]

Le planificateur topologique opère par une recherche de chemin dans le graphe orienté de cellules afin de déterminer les ressources requises à l'exécution de la trajectoire. Si un obstacle obstrue le passage, les ressources sélectionnées peuvent s'étendre aux cellules adjacentes à la voie considérée (voie "d'en face").

Le planificateur géométrique calcule des chemins non-holonomes, avec rayon de giration borné, entre deux configurations, en se restreignant aux ressources imparties et en coutournant les obstacles modélisés. L'algorithme est une recherche complète à une résolution donnée (paramétrable) de l'espace des configurations (x, y, θ), et en considérant un robot modélisé par un rectangle augmenté des incertitudes en position. Le chemin obtenu est de type "Reeds & Shepp" (voir [Latombe 91, Laumond 94]).

3. Les différents algorithmes ont été développés par L. Aguilar, R. Alami et N. Simeon.

Le séquenceur multi-robots détermine le long de la trajectoire les positions pour lesquelles le robot doit a) mettre à jour l'ensemble des ressources qu'il occupe (libération ou occupation), b) émettre un événement de synchronisation auprès d'un autre robot, c) s'arrêter et attendre un événement de synchronisation. Les figures 4.5 et 4.4 illustrent respectivement une synchronisation sur trajectoire et une synchronisation sur ressources.

Ces chemins topologiques et géométriques s'obtiennent au moyens des requêtes suivantes:

1. MP_NOMINAL_MOTION: entre deux positions nominales (positions d'entrée ou de sortie de station ou de voie). Les trajectoires nominales ne sont calculées qu'une fois et mémorisées afin de gagner du temps à l'exécution.
2. MP_SPECIFIC_MOTION: entre deux positions quelconques en ne considérant que les obstacles du modèle de l'environnement.
3. MP_AVOIDANCE_MOTION: entre deux positions quelconques en intégrant les obstacles locaux détectés par le robot et modélisés par le module PER qui les rend accessibles via le poster OBSTACLES_PERCUS.
4. MP_EXTENDED_MOTION: entre deux positions quelconques avec obstacles locaux qui nécessitent l'utilisation de ressources spatiales complémentaires.

Lorsqu'il s'avère que le robot doit partager des ressources avec d'autres robots, alors ces requêtes de planification devront également procéder à la synchronisation multi-robots. Un des paramètres de la requête permet de selectionner cette procédure. En ce cas, les chemins des autres robots sont transmis au module MP via le poster PLANS_MULTI_ROBOT (voir la figure 4.7 page 135).

MP_INIT	(E)	Initialisation
MP_NOMINAL_MOTION	(E)	Chemin entre deux positions nominales
MP_SPECIFIC_MOTION	(E)	Chemin entre deux positions quelconques
MP_AVOIDANCE_MOTION	(E)	*Idem* avec obstacles "locaux"
MP_EXTENDED_MOTION	(E)	*Idem* en s'étendant aux cellules adjacentes
MP_OCCUPIED_TRAJ_RESOURCES	(E)	Ressources occupées pour une position
MP_TRAJ_RESOURCES	(E)	Ressources qui connectent deux positions
MP_EXTENDED_TRAJ_RESOURCES	(E)	*Idem* en prenant en compte les obstacles
MP_SET_TRAJ_LENGTH_LIMIT	(C)	Pour limiter la longueur d'une trajectoire

4.2.6 Le module MON

La synchronisation étant largement basée sur l'attribution et la libération de ressources spatiales, le superviseur a besoin de connaître les ressources utilisées par le robot, et plus particulièrement les transitions marquant l'entrée (MON_ENTER_RESOURCE) et la sortie (MON_LEAVE_RESOURCE) de ressource, ou les états d'occupation d'une ressource (totale: MON_INSIDE_RESOURCE ou partielle: MON_NOT_INSIDE_RESOURCE).

Ces surveillances correspondent à des activités périodiques. En cas de surveillances identiques (*e.g.* entrée réitérée d'une même ressource), la seconde instance est maintenue dans l'état **start**. Lorsque la condition surveillée est vérifiée, l'activité courante se termine, et l'éventuelle surveillance en attente peut transiter dans la boucle principale de surveillance **exec**.

Une activité permanente calcule la position des points des polygones représentant le robot à sa taille réelle (test d'attribution de cellule) et le robot grossi de l'incertitude sur sa position (test de libération de cellule).

MON_INIT	(E)	Initialisation
MON_ENTER_RESOURCE	(nE)	Surveille l'entrée d'une ressource
MON_LEAVE_RESOURCE	(nE)	Surveille la sortie d'une ressource
MON_INSIDE_RESOURCE	(nE)	Surveille l'inclusion dans une ressource
MON_NOT_INSIDE_RESOURCE	(nE)	Surveille l'inclusion partielle d'une ressource
MON_KILL	(C)	Interruption d'une surveillance
MON_KILL_ALL	(C)	Interruption de toutes les surveillances
MON_OCCUPIED_RESOURCES	(E)	Liste des ressources occupées
MON_MONITOR_ALL_RESOURCES	(E)	Surveille toutes les entrées/sorties

4.2.7 Le module CSC

Le module CSC a la charge de transmettre des messages entre le robot et la station centrale. La réception de messages en provenance de la station centrale (CSC_RECV) est intégrée sous la forme d'une surveillance qui se déclenche si un message est présent et qui retourne le contenu du message. Pour être informé au plus tôt, le superviseur devra donc réactiver cette activité après chaque réception de message.

CSC_INIT	(E)	Initialisation
CSC_SEND	(nE)	Emission messages vers la station centrale
CSC_RECV	(E)	Attente reception message

4.2.8 Le module IRC

Le module IRC est similaire au module CSC si ce n'est qu'il est chargé du transfert de messages entre les robots. La requête IRC_SEND permet d'émettre un message auprès d'un robot donné, et la requête IRC_BROADCAST permet de transmettre un message à l'ensemble des robots.

IRC_INIT	(E)	Initialisation
IRC_SEND	(nE)	Emission messages vers un robot
IRC_BROADCAST	(nE)	Emission messages à la cantonade
IRC_RECV	(E)	Attente réception message de robot
IRC_DETECT	(E)	Identification des robots à proximité

4.3 Les interactions entre les modules et le superviseur

G^{en}oM produit avec chaque module les KA (des procédures PRS) qui permettent d'émettre les requêtes, de réceptionner les répliques et d'accéder aux posters. Par un mécanisme de scrutation, qui se déroule à la fréquence de la boucle principale d'interprétation, PRS réceptionne les répliques en temps borné et produit un nouveau fait dans la base de faits qui sera pris en compte par les mécanismes standards du noyau de PRS (voir §I.1.5 et la figure I. 1.4 page 12). Cela permet au superviseur de réagir aux événements (les répliques) et données (paramètres de répliques et posters) en provenance des modules au même titre qu'aux données produites par le déroulement des KAs. Cette mécanique permet de disposer au niveau de la base de

faits de PRS d'une image de l'exécution de la couche fonctionnelle. Chaque modification de cette image peut ainsi être directement prise en compte au niveau de PRS.

4.4 Les interfaces graphiques

Un serveur graphique 3D (bibliothèque GL sur Silicon Graphics) permet de visualiser les mouvements et les opérations de chargement ou de déchargement des robots dans leur environnement et les obstacles "imprévus" ajoutés au cours de l'expérimentation. Il réceptionne et affiche les positions, les trajectoires, les obstacles perçus par les robots et directement transmis par les modules. Il permet également de mémoriser ces informations afin de rejouer des expérimentations simulées ou réelles (en ce cas les données sont transmises via les modems). Ainsi les figures 4.3 page 130, 4.4 page 132, 4.9 page suivante et 4.14 page 146 sont des copies d'écran de cet interface pour différents environnements. Une version 2D de l'affichage (bibliothèques MOTIF et Xlib sur SunOS, Solaris, Silicon, X36) qui permet de mieux analyser les différentes situations a également été conçue (figures 4.5 page 132 et 4.15 page 146).

4.5 Une simulation complète et réaliste

Afin de pouvoir procéder à des tests réalistes et intensifs concernant la coopération multi-robots, une version UNIX du superviseur et des modules a été mise en place. Chaque robot simulé s'exécute sur une station de travail UNIX distincte; les communications transitent par Ethernet (figure 4.8). Les codes des modules et du superviseur des robots sont identiques à ceux qui seront embarqués sur les vrais robots et qui s'exécuteront sur VxWorks.

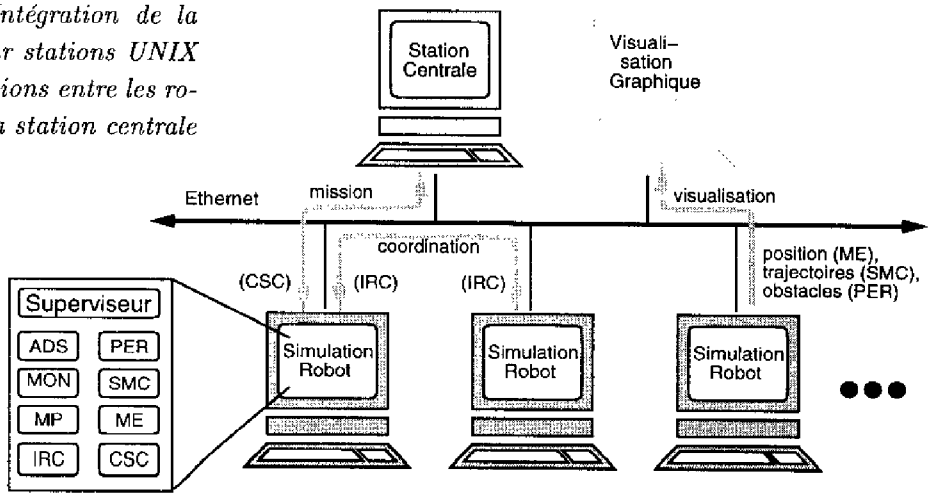


FIG. 4.8 – *Intégration de la simulation sur stations UNIX et les interactions entre les robots et avec la station centrale via Ethernet.*

Les mouvements du robot, la détection d'obstacles et les communications radio sont simulés au niveau des modules[4]. Nous avons ainsi pu exécuter de très nombreuses simulations

4. La simulation s'opère au niveau des modules MARTHA et non pas au niveau du robot logique, car nous craignions une surcharge des machines UNIX qui ne sont pas adaptées aux applications temps réel. La simulation complète (avec tous les modules de la figure 4.12 page 144) a par contre été largement employée sur les racks VxWorks pour mettre au point les modules embarqués sur les robots.

sur des environnements variés, avec des robots de caractéristiques diverses (dimensions, vitesses, rayon de giration), et faisant coopérer jusqu'à 25 véhicules (limitation dûe au nombre de stations de travail!), ce qui ce traduit par près de 500 processus s'exécutant en parallèle. Il est également intéressant de noter que par le fait de la charge variable des machines (d'autres utilisateurs les exploitent pour d'autres applications) et du réseau Ethernet, chaque expérimentation est unique et vient donc encore valider le protocole de coopération.

On a ainsi pu procéder à des centaines d'insertions de plans et le planificateur de trajectoire (le module MP) s'est révélé suffisamment efficace pour élaborer et fusionner des plans sans que les robots marquent d'arrêts intempestifs. Pour une expérimentation à 10 robots sur l'environnement de la figure 4.3 page 130, environ 400 messages ont été échangés entre les robots dont près de la moitié concerne la diffusion (broadcast) des ressources requises, un petit quart correspondant à des échanges de plans, un autre à des événements de synchronisation, et les messages restants concernant des résolutions de conflits dûs à des prises de jeton simultanées pour procéder à des OIP ou à des mises à jour des graphes de détection des inter-blocages (deadlocks). Environ 40 k-octets ont été échangés en 15mn ce qui montre que ce protocole est compatible avec des medium de communications à faible débit; cela sera confirmé par l'expérimentation réelle décrite dans la section suivante.

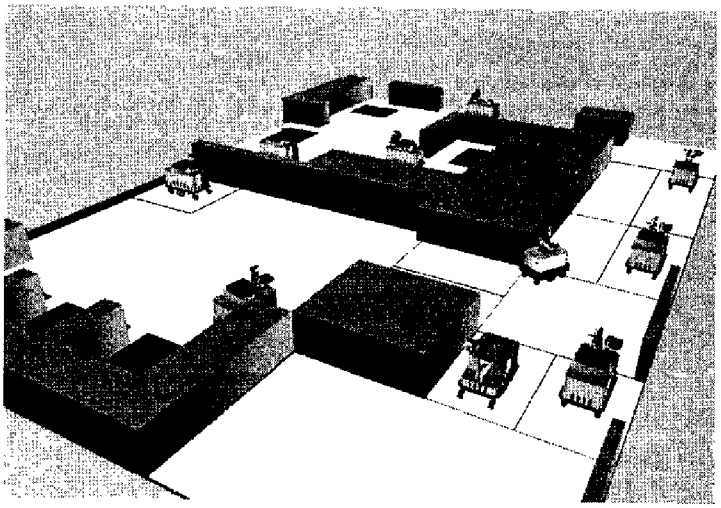


FIG. 4.9 - *Un modèle d'environnement avec 10 robots simulés.*

4.6 Une expérimentation à trois robots

Le protocole de coordination et l'ensemble des fonctions requises ont été embarqués sur les trois robots de la famille *Hilare* en service et ont donné lieu à de nombreuses expérimentations que nous allons maintenant décrire. Cette intégration à mis en œuvre l'ensemble des modules qui ont été présentés dans le chapitre III.1 concernant la couche fonctionnelle d'*Hilare2*.

4.6.1 Les robots

Les trois acteurs sont des robots dont les chassis ont été construits par la société Midi-Robot en collaboration avec le LAAS. On ne présente plus *Hilare2* qui a été largement décrit dans le premier chapitre de cette partie.

▷ **Le robot** *H2bis* trés récent est de conception identique. Cependant, présent depuis quelques mois seulement au laboratoire, il n'a pas pu être entièrement instrumenté. En particulier les ultrasons ne sont pas encore exploitables et il ne dispose ni de gyroscope, ni de télémètre laser. Ce dernier, dont la partie mécanique a été conçue et développée dans le cadre d'une collaboration LAAS-INSA, devrait être prochainement installé. Il sera beaucoup rapide que celui d'*Hilare2* (de 100Hz à 2000Hz selon la précision des tirs) et devrait permettre des localisations en mouvement[5]. Pour pallier à ce manque, nous recalons le robot à l'aide des caméras externes et du module LOCEXT (voir le chapitre III.1). Ce robot sera prochainement équipé d'un bras manipulateur 6 axes GT-Robotique/Robot-Soft de 35kg prolongé d'une pince asservie qui nous permettra d'élargir l'expérimentation STRADA à des applications de manipulation. En ce qui concerne la structure informatique, elle est équivalente à celle d'*Hilare2* si ce n'est qu'il est équipé d'un seul rack arrière hébergeant six cartes MVME 162. Les cartes de commande des capteurs et des actionneurs sont cependant de modèles plus récents ce qui a nécessité la réécriture des fonctions de la bibliothèque du robot logique (assurée par G. Bauzil et M. Khatib). Par contre nous avons pu alors profiter de cette notion de robot logique pour porter directement les modules logiciels et le superviseur d'exécution préalablement mis au point sur *Hilare2*, rendant ainsi *H2bis* opérationnel en un laps de temps très réduit.

▷ **Le robot** *Junior* est de conception plus ancienne (1985, voir [Bauzil 88]). Son électronique ne supporte pas le système d'exploitation VxWorks. L'absence de noyau temps réel, la lenteur des processeurs et la rigidité de l'architecture nous ont conduit à déporter les nouveaux modules et le superviseur du robot sur une station UNIX. Les modules de plus bas niveau qui interagissent directement avec les actionneurs et les capteurs sont bien évidemment maintenus sur le robot: les asservissements du mouvement, les acquisitions laser et ultrasonique (voir [Noreils 89]). Nous avons cependant identifier lors d'expérimentations intensives, un certain nombre de bogues au niveau de ces couches basses. Cela présente au moins l'avantage de soumettre les procédures de reprise d'erreur à des tests sévères: les défaillances vont être détectés par les modules et repris automatiquement par le superviseur.

Les trois robots sont équipés de modems radio à 9600 bauds (environ 900 octets/sec) et du protocole de transfert slip (Serial Link IP). La connection TCP-IP se fait point à point: la communication entre les trois robots et la station centrale est donc organisée en étoile. Ces modems sont le seul lien avec la station centrale durant les missions.

Les figures 4.10 et 4.11 montrent respectivement les trois robots et le contexte expérimental.

4.6.2 La couche fonctionnelle

La couche fonctionnelle comporte les modules présentés dans le chapitre III.1 auxquels ont été ajoutés les modules MARTHA. Parmi ceux-ci, les modules MP, MON, CSC et IRC ne dépendent pas d'autres serveurs et sont strictement ceux utilisés dans le cadre de la simulation sur UNIX. Par contre, les modules ME, SMC, PER et ADS sont clients des modules de base des robots. L'ensemble des modules embarqués sur les robots *Hilare2* et *H2bis* et leurs relations client/serveur sont représentés sur la figure 4.12 page 144 (afin de ne pas alourdir la figure,

5. Les données proximétriques, plus ponctuelles et donc plus précises que celles issues des ultrasons, pourront aussi être utilisées par le module AVOID pour procéder aux évitements dynamiques d'obstacles.

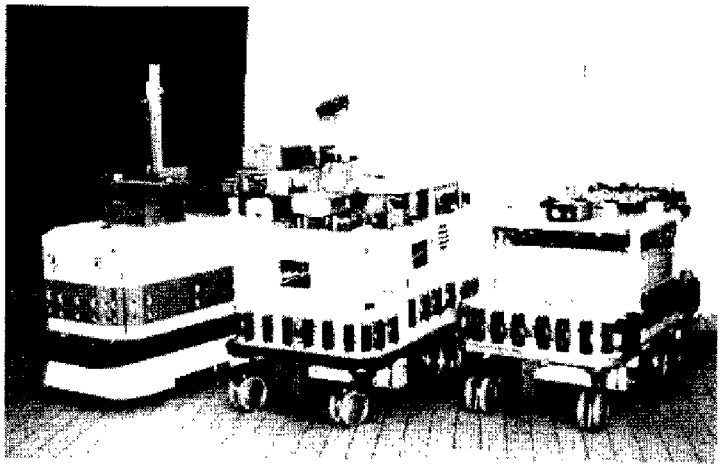


FIG. 4.10 – *Les trois robots Hilare de l'expérimentation STRADA. De gauche à droite: Junior, Hilare2 et H2bis*

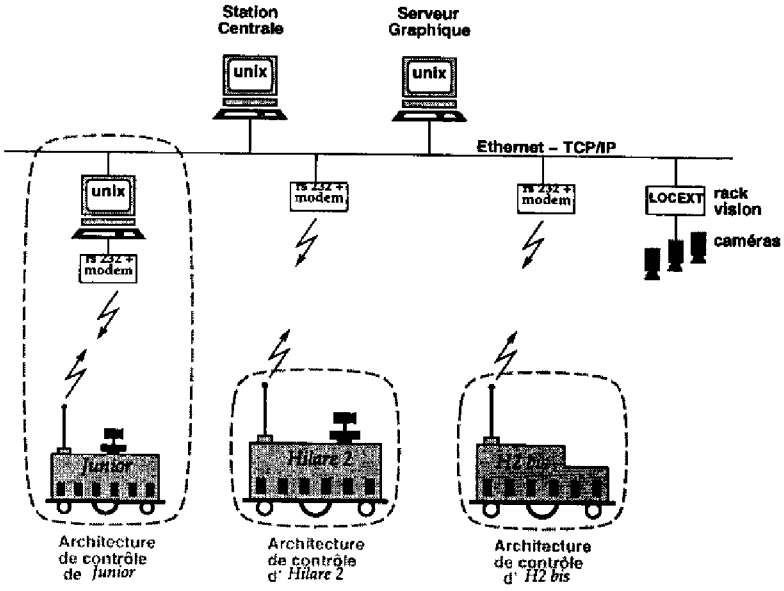


FIG. 4.11 – *Le contexte expérimental.*

les posters et les flux de données ne sont pas représentés).

Ainsi, le module ME transmet les trajectoires et les surveillances d'abscisse curviligne au module PILO qui se chargera de lisser les trajectoires (trajectoires de Reeds&Shepp lissées par des clothoïdes) et de générer les points de consigne. Il maintient également à jour le poster POSITION à partir du poster locoRobot du module LOCO (voir le III.chapitre 1).

Le module SMC s'adresse également au module PILO dont il active le mode d'évitement par la requête piloAvoidOnOff. Les consignes générées par PILO sont alors filtrées par le module AVOID qui en assure la transmission au module LOCO. Lorsque des zones limites d'évolution sont imposées au module SMC (requête SMC_SET_BOUNDARIES), celles-ci sont transmises au module US_VIR.

Le module PER requiert les acquisitions d'images laser auprès du module TELE3D, puis s'adresse au module LOCA2D pour procéder à la segmentation de ces images afin de modéliser les obstacles perçus, ou encore pour recaler le robot sur le modèle des amers. Il transmet au module LOCO la position recalée. Le robot *H2bis* ne disposant pas de télémètre laser, la

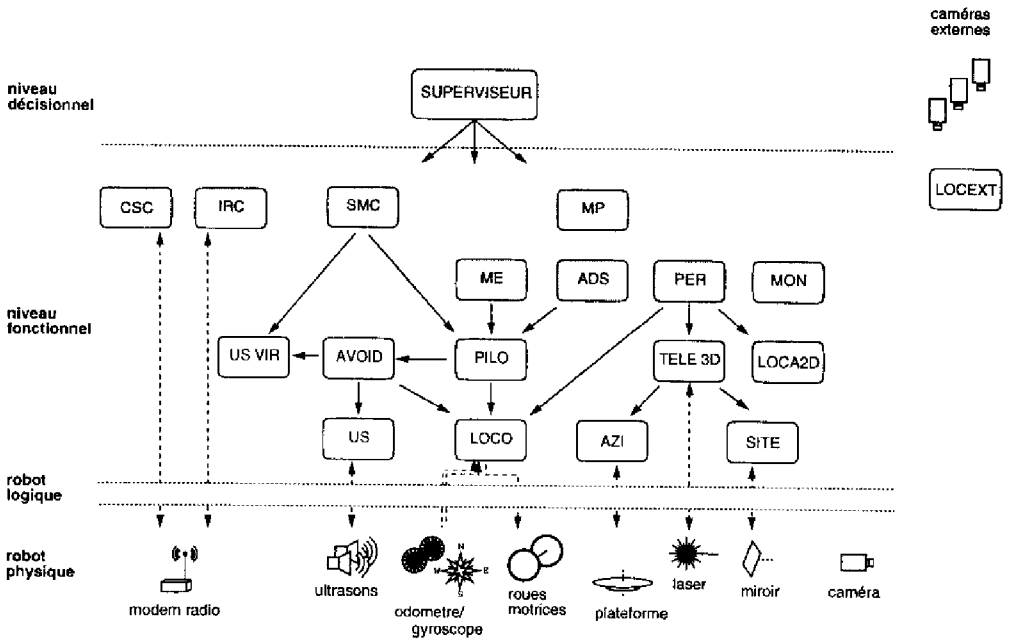


FIG. 4.12 – *La couche fonctionnelle des robots. Les modules MARTHA sont grisés.*

localisation s'effectue au moyen de caméras externes grâce au module externe LOCEXT.

Le module ADS a actuellement un rôle très limité dû à l'absence de bras manipulateur (un bras devrait être prochainement installé sur *H2bis*). Cependant, un module de base nommé DOCK, qui procède à des manœuvres très contraintes de stationnement, et faisant intervenir des boucles de modélisation local/planification, est en cours d'intégration.

4.6.3 Résultats expérimentaux

La salle robotique propose une zone d'évolution d'environ $10 \times 7\text{m}^2$ (voir le plan de la figure 4.13 page suivante), ce qui représente un environnement extrêmement contraint pour trois robots et va engendrer de nombreux conflits de ressources. Un des environnements contruit dans cette salle peut être visualisé sur les figures 4.14 page 146 et 4.15 page 146: il comporte deux aires qui incluent respectivement six stations et une station, et qui sont jointes par deux voies de sens opposé composées chacune d'une cellule unique.

Les robots initialisent tout d'abord leur position à l'aide du télémètre laser ou des caméras externes. Les missions consistent alors à se déplacer d'une station à l'autre en se coordonnant et en se recalant de temps à autre. Le robot *Hilare* est perpétuellement en mode d'évitement d'obstacles sur les données ultrasons. Si la présence d'un obstacle entraîne des écarts conséquents à la trajectoire, ce qui dans cet environnement est rapidement la cas, alors le robot s'arrête, casse son plan et procède à une modélisation des obstacles avoisinants par l'intermédiaire du laser. Une nouvelle planification de trajectoire qui intègre ces obstacles est alors exécutée. Insistons sur le fait que les autres robots ne sont pas des obstacles imprévus.

Durant les opérations de reprise suite à un problème détecté par la couche fonctionnelle et signalé au superviseur (choc arceau de sécurité, déviation importante, asservissements de *Junior* incohérents dûs à des bogues anciens dans les couches basses), les autres robots restent

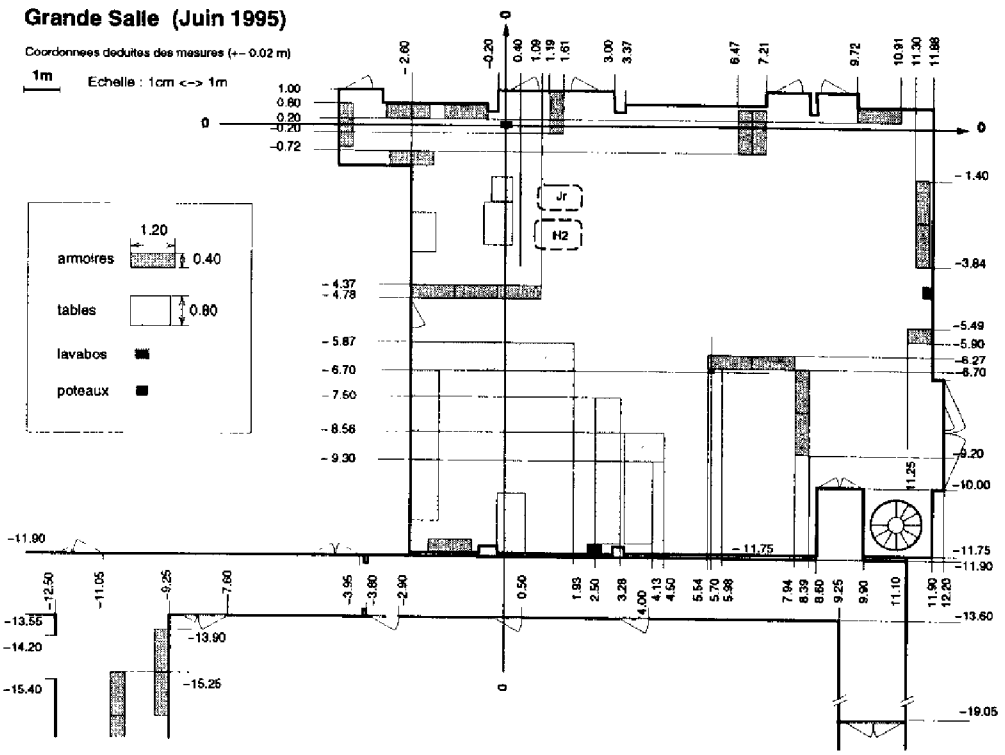


FIG. 4.13 – *Le plan de la salle robotique et du couloir d'accès.*

informés par le superviseur de la position du robot défaillant qui est alors considéré comme un obstacle.

On a pu assister à de très belles synchronisations à deux ou trois robots (j'évoque ici volontairement une attitude passive, car lors de ces expérimentations les robots sont réellement autonomes et l'opérateur a rarement à intervenir si ce n'est pour attribuer de nouvelles missions). Lors des quelques dizaines d'expérimentations nous n'avons eu à établir que trois constats de collision entre robots dûes à des erreurs de manipulation.

Voici quelques résulats quantitatifs issus d'une expérimentation d'un peu moins de 2 heures dans l'environnement des figures 4.14 et 4.15.

- Chaque robot a parcouru environ 600 m (dans un environnement contraint de 10×7 m^2 !).
- Ils se sont échangés par radio 1800 messages.
- Ils ont procédé à plus de 500 coordinations qui se sont traduites par:
  - 125 synchronisations sur trajectoires,
  - 35 synchronisations sur ressources.
- A bord de chaque robot plus de 3000 requêtes ont été échangées entre le superviseur et les modules, auxquelles il faut ajouter les requêtes émises entre les modules (nombre non-mesuré certainement du même ordre).

Le nombre important de coordinations est dû au fait que les robots partagent quasiment en permanence une même ressource (l'aire 0), qui de plus est de dimension très réduite (7×6 m^2).

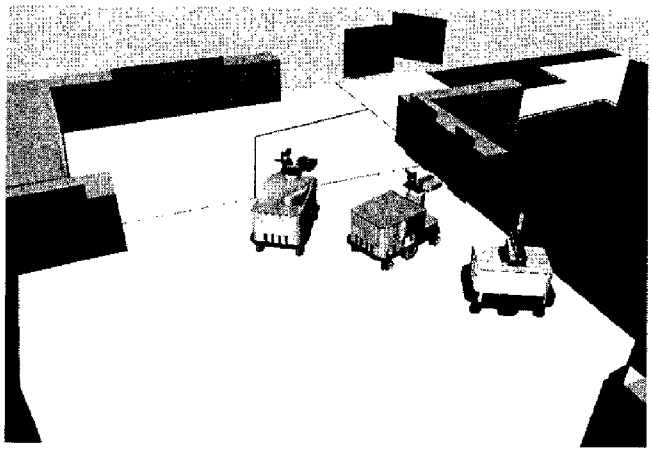
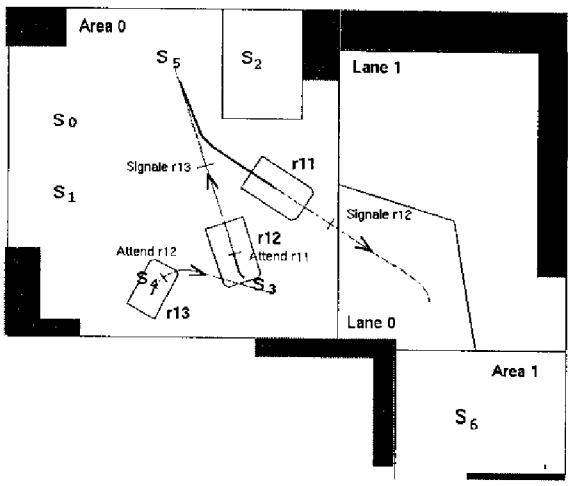


FIG. 4.14 – *Les trois robots Hilare dans la salle de robotique vus par le serveur graphique 3D. Les panneaux foncés sont des murs segmentés à partir de points lasers.*

FIG. 4.15 – *La même scène selon le graphique 2D avec les trajectoires planifiées. On y a ajouté les directions de mouvements et les points de synchronisation. (r11: Hilare, r12: H2bis, r13: Junior)*



Conclusion

▷ **Bilan** Le travail présenté dans ce mémoire contribue à la formalisation de la couche fonctionnelle de robots mobiles autonomes qui s'intègre dans une architecture de contrôle décisionnelle et qui considère les contraintes relatives aux systèmes réactifs distribués.

Cette étude nous a permis d'élaborer un langage de spécification de modules associé à un générateur, G^{en}oM, qui produit automatiquement le code correspondant à la structure de contrôle et aux interfaces des modules.

Ce générateur offre au programmeur un contexte de programmation simple et complet qui lui permet de se focaliser sur l'algorithmique et de s'abstraire des problèmes relatifs au développement de tâches temps réel et à l'intégration des services et des protocoles de communication dans l'architecture de contrôle. En outre, la production automatique des modules garantit un comportement standard et évite de procéder à des tests fastidieux concernant la logique et la réaction des modules aux requêtes. Des outils de mise au point permettent également de vérifier la satisfaction de contraintes temporelles et de guider la répartition des modules sur les différents processeurs de la machine. Enfin, les bibliothèques d'interface et la description formelle des modules (compatibilité des requêtes, ressources utilisées, bilans d'exécution) permettent d'insérer les services offerts par le module dans un exécutif.

Cette approche a été utilisée dans le cadre de différents projets (exploration planétaire, transitique) et validée par une simulation réaliste à quinze robots et surtout par deux expérimentations réalisées au LAAS sur le robot ADAM et les trois robots de la famille *Hilare*.

Ces projets, auxquels ont collaboré une dizaine de personnes, ont été grandement facilités par la production des descriptions formelles qui constituent un document complet concernant les interactions et les services offerts par chaque module. A partir de cette description exhaustive et structurée, des modules exécutables sur UNIX et VxWorks sont produits et utilisables avant même la mise au point des algorithmes.

Une vingtaine de modules ont ainsi été développés et intégrés sur les quatre robots réels et en simulation. Outre le développement de G^{en}oM et l'intégration des modules, nous avons conçu des algorithmes de génération et d'exécution de trajectoires non holonomes, ainsi qu'une méthode de localisation absolue par caméras externes.

Enfin, le concept de modules ainsi que son fonctionnement interne et ses interfaces ont été utilisés par les partenaires industriels pour la réalisation des démonstrateurs du projet MARTHA (Commutor à la SNCF et un robot Indumat à l'aéroport de Francfort).

▷ **Perspectives** L'utilisation de G^{en}oM pour nos robots mobiles est maintenant généralisée. Deux applications concernant un bras manipulateur sur station fixe et un bras manipulateur sur le robot *H2bis* pour la saisie de pièces par asservissement visuel statique et dynamique sont en cours d'étude. Par ailleurs, nous envisageons l'extension de l'outil à des applications non-robotiques qui requièrent la conception et la mise en œuvre de systèmes distribués réactifs (aéronautique, ferroviaire, spatial, automobile).

Concernant l'outil lui-même, son utilisation intensive nous a permis de dégager quelques évolutions nécessaires qui apparaîtront dans une prochaine version. Le premier point concerne la généralisation du graphe d'exécution des activités à un nombre quelconque d'états (**exec.i**).

Bien que cela ne présente pas de difficulté en soi, il faudra cependant d'un point de vue théorique démontrer que la logique interne d'un module, qui était jusqu'à présent relativement simple, reste valide. Le second point est d'offrir la possibilité de produire des événements asynchrones durant l'exécution d'une activité. Cela permettrait d'informer de l'évolution de l'état de cette activité et en particulier des différentes phases de production de données ou d'interaction avec d'autres serveurs. Il s'agit en quelque sorte de généraliser le nombre de répliques intermédiaires.

Un autre aspect important est la gestion des bilans d'exécution. En effet, dans le type d'applications que nous considérons, les incertitudes et la non-maîtrise de l'évolution du monde rendent les situations d'exécution non-nominales fréquentes. Il est dans ce contexte crucial de les détecter et surtout de les interpréter correctement afin de prendre les décisions de reprise adéquates. Actuellement chaque service retourne un bilan qui caractérise l'éventuel dysfonctionnement. Cependant la hiérarchie dynamique des activités pose le problème de la répercution en cascade d'un dysfonctionnement. En l'état actuel, le système ne n'offre pas de méthode systématique[6] pour "remonter" un bilan dont l'origine proviendrait d'un serveur du module. Il nous faut donc envisager de retourner non pas un bilan unique mais une séquence de bilans qui serait automatiquement augmentée par les clients successifs jusqu'à l'exécutif. Celui-ci disposerait alors de toutes les informations nécessaires à l'analyse du problème, de son origine et des modules impliqués. Il parait également indispensable de classifier les bilans selon leurs conséquences sur le système (module en état de marche, service hors d'usage, reprise possible avec ou sans intervention de l'opérateur) et le cas échéant de permettre la mise en place de procédures de reprise.

Une ouverture de plus grande ampleur qui nous semble être un prolongement fondamental et logique de ce travail serait de considérer non plus uniquement les modules pris séparément, mais l'ensemble des modules qui interviennent dans une application donnée. Cela aura des répercutions dans l'aide à la conception, à l'intégration et à la mise en œuvre de l'application, avec différents niveaux d'importance et de difficulté sous-jacente:

- aide à la répartition des modules sur les processeurs de la machine hôte selon les ressources CPU et les contraintes temporelles;
- spécification des contraintes entre les services des différents modules de la couche fonctionnelle et, spécification des flux de contrôle et de données pour l'application considérée;
- démonstration de propriétés logique et temporelle de l'ensemble de l'application;
- production automatique des règles qui régissent le contrôleur d'exécution (l'exécutif) de l'application. Cette la génération automatisée de l'exécutif tirerait parti des descriptions formelles des modules, et en particulier de la déclaration des compatibilités entre les requêtes, des ressources critiques utilisées et des bilans d'exécution. Elle devrait également pouvoir intégrer des contraintes spécifiques de l'application.

Ces deux derniers points rejoignent les études menées par le groupe relatives à la conception du niveau décisionnel de notre architecture, et qui font appel aux techniques de l'intelligence artificielle.

6. La technique actuelle consiste à définir un bilan spécifique pour chaque module qui signifie la défaillance d'un serveur, le bilan original transmis par ce dernier est conservé dans le poster de contrôle.

Annexes

Annexe A

Grammaire de G^{en}$_{o}$M

La grammaire comporte des symboles terminaux non définis: *identificateur*, *nom-typedef*, *expression*, *variable-c*, *référence-sdi* et des terminaux donnés littéralement imprimés en style `courier`. Cette grammaire a été simplifiée pour des raisons de lisibilité, sa version originale est acceptée par le générateur d'analyseur YACC. A cela s'ajoute la grammaire du préprocesseur (`#include, #define, #if, ...`).

unité-de-traduction ⟶
  déclaration-de-module
  | *déclaration-de-typedef*;...
  | *déclaration-de-requête*;...
  | *déclaration-de-poster*;...
  | *déclaration-de-tâche-d'exécution*;...

déclaration-de-module ⟶
  `module` *identificateur*
    { *attributs-de-module*;... }

attributs-de-module ⟶
  `internal_data:` *nom-typedef*
  | `number:` *expression*
  | `max_rqst_size:` *expression*
  | `max_reply_size:` *expression*

nom-typedef ⟶
  nom d'un type "C" prédéfini ou pas

déclaration-de-typedef ⟶
  identique à une déclaration de typedef "C"

déclaration-de-requête ⟶
  `request` *identificateur*
    { *attributs-de-requête*;... }

attributs-de-requête ⟶
  `type: control`
  | `input:` *variable-c* :: *référence-sdi*
  | `output:` *variable-c* :: *référence-sdi*
  | `c_control_func:` *identificateur*
  | `incompatible_with: none` | `all` | *identificateur*,...

| fail_msg: *identificateur,...*
| rqst_num: *expression*

attributs-de-requête ⟶
type: exec
| input: *variable-c* :: *référence-sdi*
| output: *variable-c* :: *référence-sdi*
| c_control_func: *identificateur*
| c_exec_func: *identificateur*
| c_exec_func_start: *identificateur*
| c_exec_func_end: *identificateur*
| c_exec_func_inter: *identificateur*
| c_exec_func_fail: *identificateur*
| incompatible_with: none | all | *identificateur,...*
| exec_task_name: *identificateur*
| fail_msg: *identificateur,...*
| resources: *identificateur,...*
| activity: filter | server | servo_process | surveillance
| rqst_num: *expression*

variable-c ⟶
identique à une variable "C"

référence-sdi ⟶
identique à la référence à un membre d'une structure "C"

déclaration-de-poster ⟶
poster *identificateur*
{ *attributs-de-poster;...* }

attributs-de-poster ⟶
update: auto
| data: *identificateur* :: *référence-sdi,...*
| exec_task_name: *identificateur*

attributs-de-poster ⟶
update: user
| exec_task_name: *identificateur*
| type: *nom-typedef*

déclaration-de-tâche-d'exécution ⟶
exec_task *identificateur*
{ *attributs-de-tâche-d'exécution,...* }

attributs-de-tâche-d'exécution ⟶
period: none | *expression*
| delay: none | *expression*
| priority: *expression*
| stack_size: *expression*
| c_init_func: *identificateur*
| c_func: *identificateur*
| cs_client_from: *identificateur,...*
| poster_client_from: *identificateur* :: *identificateur,...*
| resources: *identificateur,...*
| fail_msg: *identificateur,...*

Annexe B

Génération d'un module par G^{en}$_{o}$M

Les fichiers générés et les prototypes des fonctions d'interaction et des codels sont présentés ici. Pour illustrer nos propos nous avons nommé:

xxx	le module	ppp	un poster
eee	une tâche d'exécution	iii, III_STR	une variable et son type
rrr	une requête	ooo, OOO_STR	une autre variable et son type
ccc	un codel	yyy	un module serveur

B.1 Organisation du répertoire d'un module

Il est conseillé d'organiser le repertoire du modules en deux sous-repertoires qui incluent respectivement les fichiers développés par l'utilisateur et les fichiers générés par G^{en}$_{o}$M comme le montre l'exemple de la figure B.1 page suivante avec le module LOCO. La description formelle est exprimée dans le fichier `loco.gen`.

Les structures de données exportées par le modules (posters et requêtes) doivent être déclarées dans des fichiers d'en-tête distincts pour pouvoir être utilisées par les clients (`locoStruct.h` dans notre exemple). Le fichier `locoConst.h` définit les valeurs par défaut qui seront utilisées à l'initialisation de la SDI/f (dimensions, vitesses, ...). Enfin on préconise de créer autant de bibliothèques de codels qu'il y a de tâches d'exécution ou de contrôle afin de ne pas partager par mégarde des variables locales entre les activités qui ne seraient alors pas protégées (les données partagées doivent toujours transiter par la SDI/f).

B.2 Génération du module

Le logiciel de génération de module prend en argument le fichier de description formelle, et éventuellement des options pour indiquer les coordonnées des serveurs (`-I`) ou des paramètres de compilation (`-D`). Ainsi, la séquence suivante génère et compile le module XXX, client potentiel du module YYY, depuis le répertoire `auto/` qui hébergera les fichiers générés (`xxx.gen` est le fichier de description formelle):

```
auto> genom ../userLib/xxx.gen -I../../yyy -DVERBOSE
auto> gnumake
```

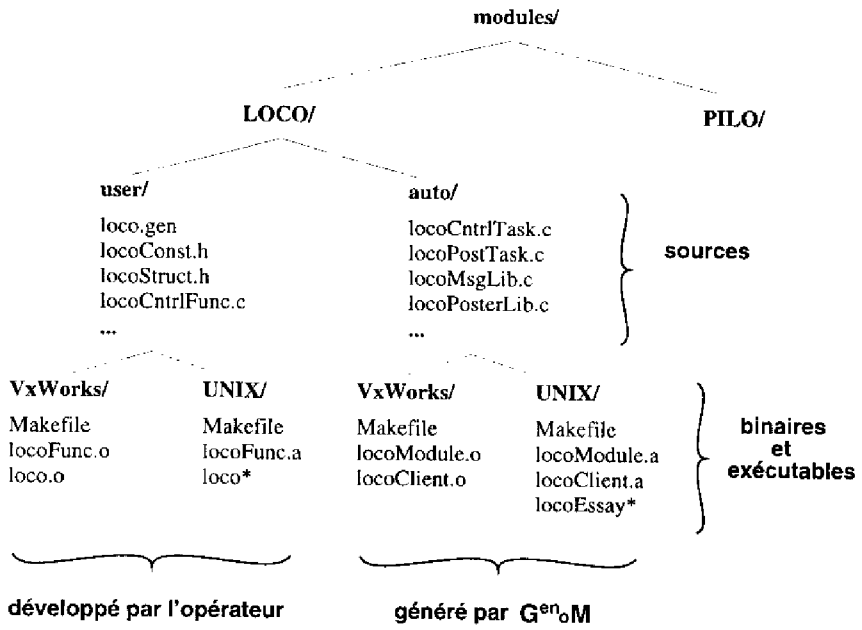


FIG. B.1 – *Exemple d'organisation pour le développement d'un module.*

Sur VxWorks le module sera chargé (`ld <loco.o`) puis lancé (`locoModuleInit`) sur une carte CPU. La tâche de test sera alors chargée (`ld <locoEssay.o`) et une ou plusieurs de ces tâches clientes du module pourront être lancées (`sp locoEssay, 1`) sur une ou plusieurs cartes CPU de la machine.

B.3 Les fichiers générés

Les fichiers sources Ces fichiers vont concerner le module, les bibliothèques d'accès au module et la tâche de test. Chacun de ces fichiers source donne lieu à la compilation à un fichier binaire, cela permet de ne charger que les bibliothèques nécessaires (par exemple le fichier `xxxPosterLib.o` si on ne compte qu'accéder aux posters du module).

	Le module	Les bibliothèques	La tâche de test

Sources	`xxxModuleInit.c`
	`xxxCmdTask.c`
	`xxxCntrlTask.c`
		`xxxMsgLib.c`
		`xxxPosterLib.c`
		`xxxPrint.c`
		`xxxScan.c`
		`xxxMsgLibPrs.c`
			`xxxEssay.c`

Binaires	`xxxModule.o`	`xxxClient.o`	
Exécutables	`xxxTaskInit`		`xxxEssay`

Les fichiers d'en-tête Les fichiers d'en-tête permettent d'accéder à des structures de données, des prototypes de fonction ou des macro-commandes. Certains concerneront uniquement

les clients et d'autre les bibliothèques des codels.

	accès au posters	clients	codels
à inclure:	xxxPosterLib.h	xxxMsgLib.h	xxxHeader.h
	xxxPosterLibProto.h	xxxMsgLibProto.h xxxMsgLibPrs.[h,kaf] xxxDeclarePrs.[h,kaf] xxxError.h	xxxType.h

B.4 Les fonctions des bibliothèques d'interaction

B.0.4 Prototypes des requêtes:

Les fonctions générées sont groupées dans la bibliothèque **xxxMsgLib**.

status *xxxRrr*RqstSend(CLIENT_ID clientId, int *pRqstId, [III_STR *iii,] int replyTimeOut);	**Emission non-bloquante de requête.** identificateur client identificateur requête argument (si défini) limite pour le temps de réponse
status *xxxRrr*ReplyRcv (CLIENT_ID clientId, int rqstId, int block, [OOO_STR *ooo,] int *activity, int *bilan);	**Réception bloquante ou non de réplique.** identificateur client identificateur requête indique si la réception est bloquante résultats (si défini) *Requête d'exécution uniquement.* identificateur de l'activité. bilan

B.0.5 Prototypes des fonctions d'accès aux posters:

Les fonctions générées sont groupées dans la bibliothèque **xxxPosterLib**.
Prototypes des fonctions d'accès aux posters d'exécution:

status *xxxPpp*PosterRead(XXX_PPP_STR *data);	**Lecture de l'ensemble du poster.** résultat de la lecture
status *xxxPpp*PosterShow();	**Affichage de l'ensemble du poster.**
status *xxxPppOoo*PosterRead(OOO_STR *ooo);	**Lecture d'un élément du poster.** résultat de la lecture
status *xxxPppOoo*PosterShow();	**Affichage d'un élément du poster.**

Prototypes des fonctions d'accès aux posters de contrôle:

status *xxx*CntrlPosterRead(XXX_CNTRL_STR *ooo);	**Lecture du poster de contrôle.** résultat de la lecture
status *xxx*CntrlPosterShow();	**Affichage du poster de contrôle.**

B.5 Les prototypes des codels

STATUS ccc (	**Codel d'une fonction de contrôle**
[III_STR *iii,]	argument (si défini)
int *bilan);	bilan retourné
ACTIVITY_EVENT ccc (	**Codel d'une fonction d'exécution**
[III_STR *iii,]	argument (si défini)
[OOO_STR *ooo,]	résultat (si défini)
int *bilan);	bilan retourné
STATUS ccc (	**Codels d'une fonction d'initialisation ou d'exécution d'une activité permanente**
int *bilan);	bilan retourné

Les codels d'exécution retournent en fin de traitement le type d'événement interne (ACTIVITY_EVENT) qui sélectionnera la transition suivante (**-/ended, -/exec.i,** ...). Tous les autres codels retournent un booléen STATUS (OK ou ERROR). Si un codel de contrôle retourne ERROR alors les arguments de la requête ne sont pas enregistrés dans la SDI/f et la réplique est retournée immédiatement. Dans le cas des codels d'initialisation ou d'exécution d'activité permanente, cela a pour effet de suspendre la tâche d'exécution.

B.6 Accès aux SDIc

Les codels peuvent avoir à accéder à des données des SDIs qui ne sont pas transmises en paramètre des codels. Pour cela, les structures des SDIs sont définies dans le fichier `xxxType.h` et les adresses de référence des SDIs sont `xxxDataStrId` pour la SDI/f et `xxxCntrlStrId` pour la SDI/c. Cependant la SDI/c doit exclusivement être accédée au moyen de fonctions définies dans le fichier `xxxHeader.h` et ses composants ne peuvent en aucun cas être altérés. Les macros-commandes d'accès sont décrites ci-dessous. Elles retournent directement la valeur attendue.

B.0.6 Les macros-commandes d'accès à la SDI/c

Accès à des informations concernant la tâche de contrôle

CNTRL_TASK_ID	identificateur de la tâche
CNTRL_TASK_STATUS	status de la tâche
CNTRL_TASK_BILAN	bilan de la tâche
NB_ACTIVITIES	nombre d'activités en cours (pour le module)

Accès à des informations concernant les tâches d'exécution L'indice i est le numéro de la tâche d'exécution.

XXX_EEE_NUM	numéro (i) de la tâche d'exécution
XXX_EEE_YYY_CLIENT_ID	identificateur de client vis à vis du serveur yyy
XXX_PPP_ID	identificateur du poster ppp
EXEC_TASK_ID(i)	identificateur de la tâche
EXEC_TASK_STATUS(i)	status de la tâche (boléen)
EXEC_TASK_PERIOD(i)	période théorique de la tâche
EXEC_TASK_ON_PERIOD(i)	durée de la dernière boucle d'exécution de la tâche
EXEC_TASK_MAX_PERIOD(i)	durée de la plus long boucle d'exécution de la tâche
EXEC_TASK_BILAN(i)	bilan de la tâche
EXEC_TASK_NB_ACTI(i)	nombre d'activité en cours (pour la tâche)
EXEC_TASK_POSTER_ID(i)	tableau des identificateurs des posters de la tâche
CURRENT_ACTIVITY_NUM(i)	numéro de l'activité en cours de traitement
EXEC_TASK_WAKE_UP_FLAG(i)	état flag d'éveil forcé de la tâche
WAKE_UP_EXEC_TASK(i)	active le flag d'éveil de la tâche

Accès à des informations concernant les activités L'indice i est le numéro de l'activité.

Informations dynamiques	
ACTIVITY_OUTPUT_ID(i)	adresse dans la SDIs des données retournées
ACTIVITY_OUTPUT_SIZE(i)	taille des données retournées
ACTIVITY_INPUT_ID(i)	adresse dans la SDIs des paramètres d'entrée
ACTIVITY_INPUT_SIZE(i)	taille des paramètres d'entrée
ACTIVITY_RQST_ID(i)	identificateur csLib de la requête
ACTIVITY_STATUS(i)	état courant de l'activité
ACTIVITY_EVN(i)	transition suivante de l'activité
ACTIVITY_BILAN(i)	bilan de l'activité
Informations statiques caractérisant la requête	
ACTIVITY_NUM(i)	numéro du type de la requête
ACTIVITY_TASK_NUM(i)	numéro de la tâche d'exécution
ACTIVITY_REENTRANCE(i)	boléen indiquant s'il peut y avoir plusieurs instances de la requête
ACTIVITY_NB_INCOMP(i)	nombre de requêtes incompatibles
ACTIVITY_TAB_INCOMP(i)	numéros des requêtes incompatibles
CNTRL_ABORT_ACTIVITY(i)	commande pour interrompre l'activité

Annexe C

Asservissement par retour d'état

S'il n'existe pas de commande par retour d'état pur pour stabiliser le robot autour d'une configuration quelconque, nous allons voir qu'une telle régulation est possible pour deux des trois paramètres de configuration du robot: soit la position (x, y) , qui garantie également une convergence en orientation. Les formalisations sont basées sur les travaux de Samson ([Samson 90]). La loi de commande qui a été retenue et intégrée sera énoncée.

Les équations cinématiques d'un mobile à deux roues motrices de rayon r et d'entraxe L sont exprimées sur la figure C:

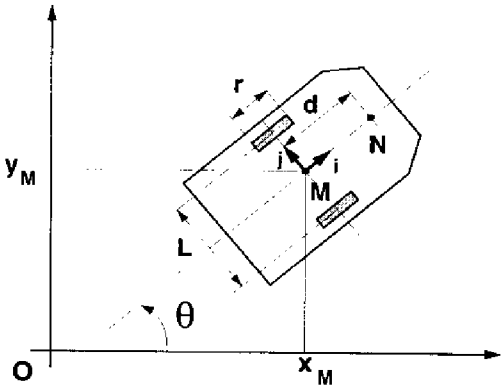


FIG. C.1 - *La cinématique d'*Hilare2

$$\begin{cases} \dot{x}_M = \frac{r}{2}\cos\theta(\dot{q}_r + \dot{q}_l) \\ \dot{y}_M = \frac{r}{2}\sin\theta(\dot{q}_r + \dot{q}_l) \\ \dot{\theta} = \frac{r}{L}(\dot{q}_r - \dot{q}_l) \end{cases}$$

(x_M, y_M) est la position du point situé au milieu de l'axe des roues dans le repère fixe $\mathcal{O}$ et θ est l'orientation du robot. q_r et q_l sont les positions angulaires des roues.

Soient v et ω les vitesses instantanées de translation et de rotation du robot. Le vecteur $(v, \omega)^t$ est lié au vecteur de commande $(\dot{q}_r, \dot{q}_l)^t$ par la matrice non-singulière:

$$\begin{pmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{r}{L} & -\frac{r}{L} \end{pmatrix}$$

On peut donc de façon équivalente raisonner sur le vecteur de commande auxiliaire $U = (v, \omega)^t$.

Exprimons le système cinématique dans le repère $\mathcal{R}$ lié au robot en considérant le cas plus général de régulation d'un point N du robot situé à une distance d de l'axe des roues ($\overrightarrow{MN} = d\vec{\imath}$). On a donc $\overrightarrow{NO} = x\vec{\imath} + y\vec{\jmath}$ et la relation vectorielle:

$$\vec{V}_{\mathcal{O}/\mathcal{R}} = -\vec{V}_{M/\mathcal{O}} - \vec{W}_{\mathcal{R}/\mathcal{O}} \wedge \overrightarrow{MO}$$

avec:

$$\begin{cases} \overrightarrow{MO} = (d+x)\vec{\imath} + y\vec{\jmath} \\ \overrightarrow{W}_{\mathcal{R}/\mathcal{O}} = \omega.\vec{k} \\ \overrightarrow{V}_{M/\mathcal{O}} = v.\vec{\imath} \end{cases}$$

On en déduit le système:

$$\dot{X} = B(X).U \tag{C.1}$$

où:

$$X = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \quad \text{et} \quad B(X) = \begin{pmatrix} -1 & y \\ 0 & -(x+d) \\ 0 & 1 \end{pmatrix}$$

En s'appuyant sur le théorème de Brockett, dans [Samson 90] il est démontré qu'il n'existe pas de loi de commande par retour d'état $U(X)$ continue qui garantisse la convergence de X vers zéro. Comme on va le voir, on peut par contre trouver une solution si l'on ne considère que deux des trois paramètres de X. On va donc chercher une loi de commande en position de la forme $U(X)$, avec $X = (x, y)^t$. On se ramène au système réduit:

$$\dot{X} = B(X).U \quad , \quad B(X) = \begin{pmatrix} -1 & y \\ 0 & -(x+d) \end{pmatrix} \tag{C.2}$$

La loi de commande $U(X) = (k_x.x, k_y.y)^t$ garantit la convergence exponentielle de X vers zéro, en effet, considérons la fonction de Lyapunov $V(X) = \frac{1}{2}\|X\|^2$. D'après le système C.2, on a :

$$\dot{V}(X) = -k_x.x^2 - k_y.d.y^2$$

et donc :

$$\dot{V}(X) \leq -\inf(k_x, k_y.d).\|X\|^2 \quad \Rightarrow \frac{\dot{V}(X)}{V(X)} \leq -2.\inf(k_x, k_y.d)$$

On peut en conclure que pour $d \neq 0$, $V(X)$, et par conséquent $\|X\|$, tend *exponentiellement* vers zéro. D'après la loi de commande $U(X)$: $\dot{\theta} = k_y.y$, la conséquence supplémentaire est la convergence de θ vers une valeur constante *à condition que d soit non-nulle.* Cela justifie notre choix de commander un point qui n'est pas situé sur l'axe des roues. Des expérimentations nous ont en effet confirmées que dans ce cas le robot avait tendance à s'enrouler autour du point de consigne.

On introduit en outre un terme d'anticipation $W = (v_r, \omega_r)^t$ qui permet de réduire l'erreur de traînage dans le cas d'un asservissement sur un point en mouvement, lorsqu'on dispose d'une estimation du mouvement de ce point. En particulier, dans la cas de trajectoires dont la dynamique est définie, telles que celles générées par le module pilotage, ce terme correspond à une commande en boucle ouverte de la trajectoire; la consigne en position permettant permet de réguler ces vitesses en fonction de l'écart en position. On remarque que cette loi de commande n'est plus une pure loi de commande par retour d'état. Enfin, pour améliorer la dynamique de la commande des termes intégrateurs ont été ajoutés. Les vitesses et leurs variations sont bornées afin de ne pas saturer les variateurs.

$$\begin{cases} v = kp_x.x + ki_x.\int x + v_r \\ \omega = kp_y.y + ki_y.\int y + \omega_r \end{cases} \quad \text{avec} \quad \begin{cases} v < v_{max} \quad \text{et} \quad \dot{v} < \alpha_{max} \\ \omega < \omega_{max} \quad \text{et} \quad \dot{\omega} < \gamma_{max} \end{cases}$$

On constate à cette occasion que pour assurer un bon suivi de trajectoire, il est primordial de ne pas saturer les vitesses et les accélérations afin de permettre la régulation. Le calcul des profiles de vitesse est assuré par le module PILO (voir l'annexe D).

Nous avons également expérimenté une loi de commande instationnaire en (x, y, θ) proposée par G. Walsh.

$$\begin{cases} v = k_1.x + y.(\omega + k_3.\sin\eta) \\ \omega = k_2.\Delta\theta + y.k_3.\cos\eta \end{cases}$$

avec: $\dot{\eta} = y$. On a pu effectivement observer la convergence du mouvement vers la configuration de consigne, cependant comme cela était prévisible, cette convergence n'est pas exponentielle. En particulier cette commande génère une succession d'oscillations pour les paramètres x et θ avec une convergence très lente en y jusqu'à la position d'équilibre.

Annexe D

Génération de trajectoires

Nous allons dans un premier temps écrire les équations cartésiennes des clothoïdes et des anticlothoïdes. Puis nous résumerons et comparerons leurs propriétés respectives. L'algorithme de calcul des profiles de vitesse optimaux de parcours de trajectoires composées de segments, d'arcs de cercles, de clothoïdes et d'anticlothoïdes sera l'objet de la troisième partie de cette annexe. Enfin, nous présenterons l'algorithme de lissage de trajectoires de type Reeds&Shepp par des clothoïdes qui a été intégré dans le module PILO sous la forme du codel **start** de la requête `piloReedsShepp`.

D.1 Equations cartésiennes des clothoïdes et des anticlothoïdes

Les équations cartésiennes se déduisent des équations intrinsèques et du système différentiel général suivant:

$$\begin{cases} dx = ds\cos\theta \\ dy = ds\sin\theta \end{cases}$$

Les clothoïdes L'équation intrinsèque d'une clothoïde est: $\kappa = k_c s$. On en déduit les équations cartésiennes suivantes:

$$\begin{cases} x(\theta) = \sqrt{\frac{\pi}{k_c}} CF(\sqrt{\frac{2\theta}{\pi}}) \\ y(\theta) = \sqrt{\frac{\pi}{k_c}} SF(\sqrt{\frac{2\theta}{\pi}}) \end{cases}$$

$CF(x)$ et $SF(x)$ sont respectivement le cosinus et le sinus de Fresnel:

$$\begin{cases} CF(x) = \int_0^x \cos\frac{\pi u^2}{2}\,du \\ SF(x) = \int_0^x \sin\frac{\pi u^2}{2}\,du \end{cases}$$

Les équations dynamiques, lorsque l'accélération linéaire est nulle et l'accélération angulaire constante, s'obtiennent en posant: $\theta(t) = \gamma t^2/2 = \frac{\alpha}{L}t^2$

La figure D.1 page suivante représente une clothoïde.

Les anticlothoïdes L'équation intrinsèque d'une anticlothoïde est: $\rho = k_a\theta$. On en déduit les équations cartésiennes suivantes:

$$\begin{cases} x(\theta) = k_a(\cos\theta + \theta\sin\theta - 1) \\ y(\theta) = k_a(\sin\theta - \theta\cos\theta) \end{cases}$$

Les équations dynamiques, lorsque l'accélération linéaire est constante et l'accélération angulaire nulle, s'obtiennent en posant: $\theta(t) = \omega_0 t$.

La figure D.2 représente une anticlothoïde. L'origine de l'anticlothoïde correspond en dynamique à un point de rebroussement: en ce point la vitesse linéaire s'annule puis change de signe.

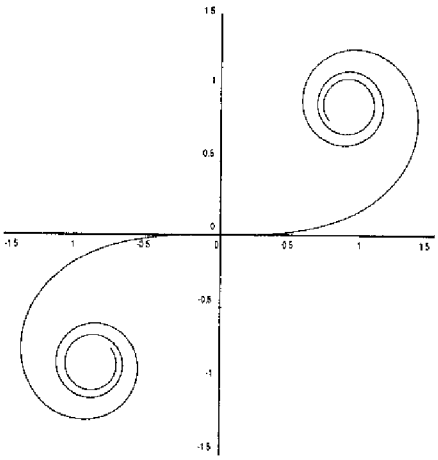


FIG. D.1 - *Une clothoïde.*

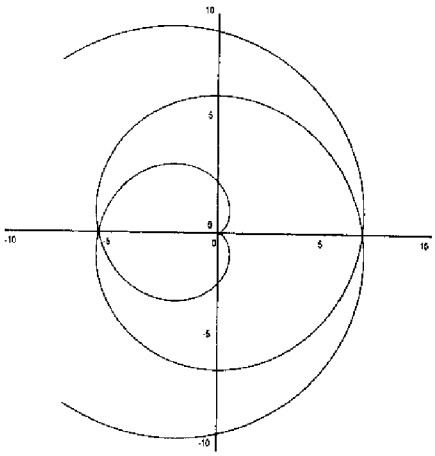


FIG. D.2 - *Une anticlothoïde.*

D.2 Pourquoi nomme-t-on une développante de cercle une anti-clothoïde ?

La dualité des clothoïdes et des anticlothoïdes est clairement exprimée par la tableau ci-dessous qui résume les propriétés respectives des deux courbes.

	clothoïdes $(u_r = -u_l)$	**anti-clothoïdes** $(u_r = u_l)$
équations intrinsèques	$\kappa = k_c s$	$\rho = k_a \theta$
accélérations	$\alpha = 0, \gamma = \gamma_0$	$\gamma = 0, \alpha = \alpha_0$
vitesses	$v = v_0$	$\omega = \omega_0$
cas dégénérés	rotation pure $k_c = \infty$, $v_0 = 0$	translation $k_a = \infty$, $w_0 = 0$
enchaînements avec:	droites $\kappa = 0$	rotation pure $\rho = 0$
applications	autoroutes	rebroussements

D.3 Calcul des profiles de vitesses optimaux d'une trajectoire

Cet algorithme intégré par un codel du module PILO permet de calculer les profiles de vitesses trapézoïdaux optimaux d'une trajectoire composée de séquences quelconques de seg-

ments, d'arcs de cercles, de clothoïdes et d'anticlothoïdes en respectant les contraintes suivantes:

- vitesses nulles au départ et à l'arrivée (cette contrainte peut être levée),
- limites sur les vitesses linéaires et angulaires,
- limites sur les accélérations linéaires et angulaires.

Ces limites peuvent avoir des valeurs différentes pour chaque type de primitive. Il par contre indispensable qu'elles soient inférieures à celles fixées par les asservissements afin de permettre la régulation.

Le calcul se base sur l'hypothèse que les clothoïdes et les anticlothoïdes se parcourent respectivement à vitesses linéaire et angulaire constantes. Cela implique que ces primitives soient enchaînées sans discontinuité de courbure (pas de point d'arrêt). En effet, cela aurait pour conséquence de les parcourir à vitesse nulle! Cette contrainte de parcour à vitesse constante pourra cependant être relaxée lors de l'exécution, en particulier pour ralentir, voir s'arrêter.

L'algorithme consiste alors à parcourir la trajectoire, en partant de vitesses nulles, et à déterminer primitive par primitive qu'elles sont les vitesses maximales atteignables au point terminal de la primitive en respectant les limitation en accélération. De plus, pour les clothoïdes, les anti-clothoïdes et les primitives se terminant par un point d'arrêt, il faut vérifier que les vitesses initiales ne conduisent pas à un dépassement de limites. En ce cas, ces vitesses initiales sont redéterminées, et à partir de ces nouvelles contraintes, les profiles de vitesses sont recalculés en parcourant la trajectoire "en marche arrière" (backtrack) jusqu'à ce que toutes les contraintes soient satisfaites. Les segments de droites et les arcs de cercles vont servir en quelques sortes de tampon car ils autorisent des accélérations et des décelérations sur une même portion.

Voici les contraintes à respecter pour une clothoïde: $v < \omega_{max}\rho_{min}$ et $v < \sqrt{\gamma/k_c}$

Voici les contraintes à respecter pour une anti-clothoïde: $\omega < v_{max}\kappa_{min}$ et $\omega < \sqrt{\alpha/k_a}$

Enfin, l'écart de vitesses maximal entre deux extrémités 0 et 1 d'un segment de longueur l est conditionné par la relation: $v_1 \leq \sqrt{{v_0}^2 + 2l\alpha}$. Pour un arc de cercle il faut ajouter la contrainte: $\omega_1 \leq \sqrt{{\omega_0}^2 + 2\theta\gamma}$.

Les profiles de vitesses ainsi déterminés fixent les vitesses et accélérations maximales tolérables sur chaque portion de la trajectoire. Ces valeurs pourront être cependant modulées à l'exécution entre 0 et cette limite supérieure(requête `piloSlow`) .

D.4 Lissage de chemin de Reeds & Shepp

Contrairement à la ligne brisée il n'y a pas dans ce cas de discontinuité de la tangente de la trajectoire mais uniquement une discontinuité de la courbure. Comme l'indique la figure 2.3 page 107 cette discontinuité peut être résorbée en enchaînant les droites et les cercles par clothoïdes, c'est ce que nous allons décrire ici. Les points de rebroussements n'ayant pas encore été considérés dans ce cas, le robot devra y marquer un arrêt.

D.0.7 Altération du chemin initial

On montre aisément *qu'il n'existe pas de clothoïde qui joigne une droite et un cercle tangent à cette droite avec une continuité d'ordre 2* (*i.e.* continuité en courbure).[1] Cette impossibilité est bien sûr également vraie dans le cas de l'enchaînement de deux cercles tangents. Par contre nous allons démontrer qu'il existe toujours une clothoïde de jonction lorsque le cercle et la droite sont strictement disjoints. Il va donc falloir, préalablement au lissage, procéder à une altération du chemin initialement fournit.

Enchaînements "segment/arc de cercle" sans rebroussement: Pour dissocier le segment du cercle sans trop altérer le chemin initial, le cercle est réduit mais conserve cependant un point de tangence avec le cercle initial. Ce point sera spécifié par la suite.

Enchaînements "arc de cercle/arc de cercle" sans rebroussement: Ce type d'enchaînement pose plus de difficultés car il n'existe pas dans le cas général de clothoïde de jonction entre deux cercles même lorsque ces cercles sont disjoints. Nous avons donc choisi de nous ramener au problème précédent en remplaçant les deux cercles consécutifs par des cercles réduits reliés par leur tangente commune. Comme précédemment les cercles réduits sont maintenus tangents à leur cercle de départ respectif. La tangente commune est choisie de façon à respecter le sens du mouvement sur chaque cercle. Dans ce cas les cercles subiront donc une double réduction.

Le point de tangence entre le cercle initial et le cercle réduit est déterminé de la façon suivante: Si l'arc de cercle marque le début ou la fin d'un mouvement (*i.e.* début ou fin de trajectoire ou point de rebroussement) alors les cercles réduits seront tangents en ce point d'arrêt. En cas contraire, on symétrise le problème en tangentant le petit cercle au milieu de l'arc du grand cercle.

Après ces réductions successives, on dispose d'un chemin qui est peu altéré par rapport au chemin initial et qui surtout est "lissable". Nous allons maintenant décrire la procédure de lissage.

D.0.8 Enchaînement droite-cercle par une clothoïde

On cherche à caractériser la clothoïde (constante k et longueur angulaire ϕ_J) qui joigne un segment et un arc de cercle de rayon ξR. Ce cercle tangente en un point V un cercle de rayon R qui est lui même tangent au segment (voir figure D.3 page ci-contre). Nous allons voir que pour chaque valeur de $\xi \in [0..1]$ il existe une et une seule solution. Nous proposons une heuristique pour choisir cette solution.

Equation de la clothoïde:

$$\begin{cases} x = kCF\sqrt{\frac{2\phi}{\pi}} \\ y = kSF\sqrt{\frac{2\phi}{\pi}} \end{cases} \qquad \text{avec: } k = \rho\sqrt{2\pi\phi} \tag{D.1}$$

Equation du cercle réduit (d'un facteur ξ) dans son repère propre:

$$\begin{cases} X = \xi R\cos\Phi \\ Y = \xi R\sin\Phi \end{cases} \qquad \text{avec: } 0 \leq \xi \leq 1 \tag{D.2}$$

1. Démonstration: Considérons une clothoïde qui tangente un cercle avec une continuité d'ordre 2. En parcourant cette clothoïde dans le sens d'accroissement du rayon de courbure (on veut joindre une droite) on s'éloigne strictement de ce cercle qu'on ne pourra donc rejoindre au point de tangence avec la droite.

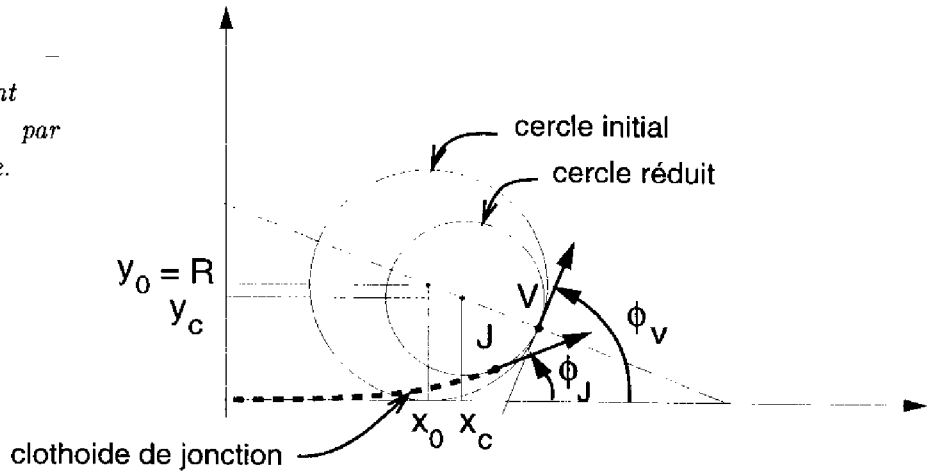


FIG. D.3 – *Enchaînement droite-cercle par une clothoïde.*

Changement de variables pour exprimer le cercle dans le repère de la clothoïde:

$$\begin{cases} \Phi = \phi - \frac{\pi}{2} \\ X = x - x_c \\ Y = y - y_c \end{cases} \tag{D.3}$$

Les coordonnées du centre du cercle réduit étant:

$$\begin{cases} x_c = x_0 + R(1-\xi)\sin\phi_V \\ y_c = R(1-(1-\xi)\cos\phi_V) \end{cases} \tag{D.4}$$

En exprimant la continuité en ordonnée ($y = y_J$), en tangence ($\phi = \phi_J$) et en rayon de courbure ($\rho = \xi R$) au point J entre la clothoïde et le cercle, on peut expliciter le coefficient de réduction du cercle en fonction de l'angle du virage et de la tangente en J:

$$\xi = \frac{1-\cos\phi_V}{\sqrt{2\pi\phi_J}SF\sqrt{\frac{2\phi_J}{\pi}} + \cos\phi_J - \cos\phi_V} \tag{D.5}$$

On en déduit l'abscisse du centre du cercle initial x_0 (et donc d'après (D.4) celle du cercle réduit):

$$x_0 = R(\xi\sqrt{2\pi\phi_J}CF\sqrt{\frac{2\phi_J}{\pi}} - \xi\sin\phi_J - (1-\xi)\sin\phi_V) \tag{D.6}$$

On obtient donc un système à trois inconnues: $\xi, \phi_J et x_0$, et deux équations: (D.5) et (D.6). Avec les contraintes suivantes:

$$\begin{cases} \phi_J & \leq & \phi_V & & \\ x_0 & \leq & l & & \\ \rho_{\min}/R & \leq & \xi & \leq & 1 \end{cases} \tag{D.7}$$

l étant la longueur du segment et $\rho_{\min}$ le rayon de giration minimal du véhicule qui est nul dans notre cas.

Heuristique pour sélectionner les paramètres Ce que l'on souhaite c'est prendre le virage le plus rapidement possible. Si l'on considère les cas extrêmes où $\xi = 1$ (les deux cercles sont confondus) et $\xi = 0$ (le cercle réduit se ramène à un point) le véhicule devra marquer un arrêt au point J. La solution se situe donc entre ces deux extrêmes.

En fait la portion de clothoïde permet de prendre de la vitesse angulaire ω entre la droite et le cercle. Cette vitesse étant bornée à $\omega_{\max}$ il est inutile de prolonger cette phase d'accélération angulaire car cela nécessiterait alors de réduire la vitesse linéaire de parcours de la clothoïde.

La vitesse angulaire maximale que l'on peut atteindre sur un arc de clothoïde étant: $\omega_{\max} = \sqrt{2\alpha}\sqrt{\phi_J}$, où α est l'accélération angulaire, on propose l'heuristique suivante: on fixe $\phi_J = \min(\alpha\omega^2{}_{max}/2, \phi_V)$ on en déduit ξ et x_0. Si x_0 est trop grand on réduit ϕ_J et on réitère la boucle précédente jusqu'à satisfaire toutes les contraintes.

Annexe E

Localisation externe

La localisation externe et absolue d'un véhicule a été présentée dans la section 2.2 sans en dévelloper les aspects calculatoires. Dans cette annexe vont être décrites successivement les techniques de calcul incrémentales de la position du motif et les méthodes de calibration des transformées constantes $T_{C/A}, T_{I/C}$ et $T_{R/M}$ (calibration du motif et de la caméra) visualisées sur la figure 2.7 page 110. Nous allons montrer en particulier que les problèmes de la localisation incrémentale du motif d'une part, et la calibration de sa position sur le robot d'autre part, conduisent à des équations non homogènes avec contraintes qui peuvent cependant être résolues par des techniques de moindres carrés car elles sont singulières.

Dans une dernière partie, nous présentons une méthode de calibration de l'odométrie qui a été élaborée à partir de la localisation externe.

E.1 Localisation incrémentale par moindres carrés

Le motif est caractérisé par des points que l'on va tour à tour identifier et localiser. Le problème consiste à déduire du modèle du motif et au fur et à mesure de l'identification de points, la configuration (x, y, θ) du motif. L'estimation de cette configuration va permettre d'identifier de nouveaux points, qui a leur tour permettront d'affiner les valeurs (x, y, θ). Le calcul devra donc être incrémentale.

Notons $T_{M/\Pi}$ la transformée cherchée, $P_{i/M} = (\ X_i \quad Y_i\ )^t$ un point du modèle et $P_{i/\Pi} = (\ x_i \quad y_i\ )^t$ le point correspondant dans la scène:

$$T_{M/\Pi} = \begin{pmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Pour chaque point apparié, nous avons donc la relation:

$$P_{i/\Pi} = T_{M/\Pi} * P_{i/M} \qquad \Leftrightarrow \qquad \begin{cases} X_i = x_i \cos\theta - y_i \sin\theta + t_x \\ Y_i = x_i \sin\theta + y_i \cos\theta + t_y \end{cases}$$

Et pour N points, nous avons un système à $2N$ équations et quatre inconnues: $t_x, t_y, \cos\theta$ et $\sin\theta$ reliées par la relation: $\cos^2\theta + \sin^2\theta = 1$. Nous allons voir que ce système peut être résolu par moindres carrés avec contraintes de façon exacte car il est *singulier*.

Posons $V = (\ t_x \quad t_y\ )^t$ et $U = (\ \cos\theta \quad \sin\theta\ )^t$ avec $\|U\| = 1$. Le système s'écrit $AU + BV = C$, où A et B sont des matrices $(2N, 2)$ et C est une matrice $(2N, 1)$ dont les lignes $2i$ et $2i+1$ sont:

$$A_i = \begin{pmatrix} x_i & -y_i \\ y_i & x_i \end{pmatrix} \qquad B_i = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad C_i = \begin{pmatrix} X_i \\ Y_i \end{pmatrix}$$

Minimiser $\|AU + BV - C\|^2$ sous la contrainte $1 - \|U\|^2 = 0$ est équivalent à minimiser le lagrangien $L = \|AU + BV - C\|^2 - \lambda.(1 - \|U\|^2)$. Les dérivée partielles devant être nulles, on obtient le système:

$$\begin{cases} A^t(AU + BV - C) - \lambda U = 0 \\ B^t(AU + BV - C) = 0 \\ \|U\|^2 = 1 \end{cases} \quad \Leftrightarrow \quad \begin{cases} V = (B^tB)^{-1}B^t(C - AU) \\ A^tDAU = \lambda U + A^tDC \\ \|U\|^2 = 1 \end{cases}$$

avec $D = Id_2 - B(B^tB)^{-1}B^t$, où Id_2 est la matrice identité.

Le système peut être résolu aisément car on peut démontrer que $A^tDA = k.Id_2$. Et par conséquent: $(k - \lambda).U = A^tDC$.

Posons $E = A^tDC$:

$$E = \begin{pmatrix} \sum(x_iX_i + y_iY_i) - \frac{1}{N}(\sum x_i \sum X_i + \sum y_i \sum Y_i) \\ \sum(x_iY_i - y_iX_i) - \frac{1}{N}(\sum x_i \sum Y_i - \sum y_i \sum X_i) \end{pmatrix}$$

E peut aisément être calculé incrémentalement. Finalement, si E n'est pas nul, le résultat cherché est:

$$U = \pm\frac{E}{\|E\|} \quad et \quad V = \frac{1}{N}\sum \begin{pmatrix} X_i - x_i\cos\theta + y_i\sin\theta \\ Y_i - x_i\sin\theta - y_i\cos\theta \end{pmatrix}$$

E.2 Calibration motif/robot par moindres carrés

La calibration de la position du motif par raport au robot n'est, a priori, pas triviale car l'origine du repère robot (milieu de l'entraxe des roues) est difficile à identifier. La localisation externe va nous donner le moyen de procéder à cette calibration de manière automatique. En effet, comme nous allons le voir, il suffit de connaître les déplacements relatifs du robot et ceux correspondants du motif, dans des repères qui peuvent être distincts pour identifier la transformation constante $T_{M/R}$. Les déplacements D_R du robot seront obtenus par l'odométrie (qui est très précise pour des petits déplacements) et les déplacements D_M du motif par la localisation externe.

Comme cela est illustré par la figure E.1 page ci-contre, les repères du motif et du robot sont liés par la relation suivante:

$$T_{M/R} * D_M = D_R * T_{M/R}$$

Avec:

$$T_{M/R} = \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix} \qquad D_R = \begin{pmatrix} R_R & T_R \\ 0 & 1 \end{pmatrix} \qquad D_M = \begin{pmatrix} R_M & T_M \\ 0 & 1 \end{pmatrix}$$

Par conséquent:

$$\begin{cases} R * R_M = R_R * R \\ R * T_M + T = R_R * T + T_R \end{cases} \tag{E.1}$$

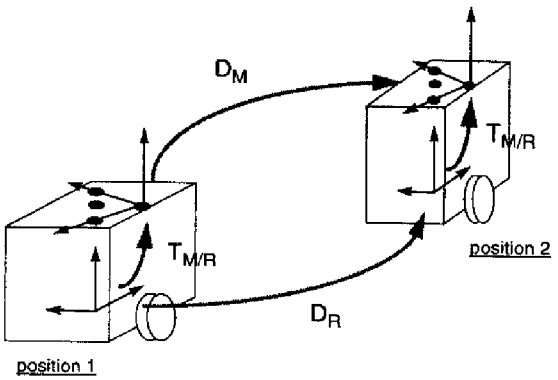


FIG. E.1 – *Relations entre le robot et le motif après un déplacement.*

La première équation de ce système permet d'identifier R si et seulement si les axes de rotation ne sont pas colinéaires. Notre robot évoluant essentiellement dans un plan horizontal, cette technique n'est pas adaptée à la détermination d'une transformation 3D. Par conséquent le motif devra être fixé de façon parfaitement horizontale sur le robot.

Posons: $D_M = (\ x_M \quad y_M\ )^t$, $D_R = (\ x_R \quad y_R\ )^t$, $T = (\ x \quad y\ )^t$, et θ et θ_R les angles des rotations R and R_R dans le plan (Oxy). La seconde équation du système E.1 permet alors d'écrire:

$$\begin{cases} x_M \cos\theta - y_M \sin\theta + x(1 - \cos\theta_R) + y \sin\theta_R = x_R \\ y_M \cos\theta + x_M \sin\theta - x \sin\theta_R + y(1 - \cos\theta_R) = y_R \end{cases} \tag{E.2}$$

Pour N mouvements, on obtient un système de $2N$ équations et quatre inconnues: $x, y, \cos\theta$ and $\sin\theta$ reliées par la relation $\cos^2\theta + \sin^2\theta = 1$. Deux mesures déplacements suffisent donc à résoudre le système. Cependant chaque déplacement supplémetaire permettra d'améliorer la précision du résultat. Une méthode très similaire à celle présentée dans la section précédente, faisant appel aux moindres carrés, peut être appliquée. En reprenant les mêmes notations, on a:

$$A_i = \begin{pmatrix} x_{M_i} & -y_{M_i} \\ y_{M_i} & x_{M_i} \end{pmatrix} \qquad B_i = \begin{pmatrix} 1 - \cos\theta_{R_i} & \sin\theta_{R_i} \\ -\sin\theta_{R_i} & 1 - \cos\theta_{R_i} \end{pmatrix} \qquad C_i = \begin{pmatrix} x_{R_i} \\ y_{R_i} \end{pmatrix}$$

Nous avons la chance une fois encore d'être en présence d'un système qui se simplifie: on peut montrer que $A^t D A = k.Id_2$. On peut donc calculer E, et finalement U et V (l'expression étant complexe, on ne la donnera pas ici).

E.3 Calibration des caméras

Pour calibrer les caméras nous avons utilisé le modèle sténopé pour lequel la matrice de transformation homogène est:

$$T_{C/I} = \begin{pmatrix} \alpha_i & 0 & i_0 & 0 \\ 0 & \alpha_j & j_0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

où α_i et α_j dépendent des distances focales et des dimensions de la barrette CCD, et (i_0, j_0) est le centre optique. La méthode de calibration employée, proposé par Tsaï, produit directement les matrices intrinsèques et extrinsèques des caméras.

La position d'un point de la scène est alors obtenue par une simple projection inverse de ses coordonnées image dans le plan Π du motif.

Cependant les caméras produisent des images plus ou moins distordues qui s'éloignent du modèle linéaire. L'adjonction de termes quadratiques permet de modéliser correctement cette distorsion (voir [Orteu 91]), mais le calcul de la projection inverse devient alors très couteux en temps. C'est pourquoi nous conservons le modèle sténopé pour la procédure de localisation/identification, mais nous envisageons d'appliquer en un seul point, le barycentre du motif, le modèle quadratique.

E.4 Calibration de l'odométrie

Nous avons développé une technique de calibration de l'odométrie, c'est-à-dire des rayons des roues et de la longueur de l'entraxe à partir de la localisation externe. Elle se base sur les écarts entre le chemin requis et exécuté par le robot et le chemin visualisé par les caméras selon le principe décrit ci-dessous:

Le vecteur d'état (position et vitesses) qui intervient dans la boucle d'asservissement du véhicule est déterminé à partir de roues odométriques de rayons R_d et R_g disposées sur l'axe des roues motrices et distantes d'une longueur L. Le point de référence (x, y) du robot se situe au milieu de cet axe, et θ est son orientation. On note $\dot{q}_d$ et $\dot{q}_g$ les vitesses angulaires des roues et v et ω les vitesses linéaire et angulaire du véhicule. L'équation de la cinématique du robot est la suivante:

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = \omega \end{cases} \quad \text{avec} \quad \begin{cases} v = \frac{1}{2}(R_d\dot{q}_d + R_g\dot{q}_g) \\ \omega = \frac{1}{L}(R_d\dot{q}_d - R_g\dot{q}_g) \end{cases} \tag{E.3}$$

La configuration du véhicule s'obtient par intégration de ce système au cours des déplacements. On constate que si les paramètres géométriques L, R_d et R_g ne sont pas estimés avec une bonne précision la configuration calculée sera entachée à chaque itération d'une erreur systématique qui va se cumuler le long de la trajectoire.

Par exemple si le rapport R_d/R_g est erroné alors le véhicule exécutera un arc de cercle en lieu et place de la ligne droite requise. En fait la procédure de calibration se base sur ce phénomène.

Notons A le rapport vrai des rayons de roues R_d/R_g, et $\hat{A}$ le rapport de l'estimée des rayons des roues $\hat{R}_d$ et $\hat{R}_g$. D'après le système E.3 le rayon de courbure d'une trajectoire s'écrit:

$$\rho = \frac{v}{\omega} = \frac{L}{2}\frac{A + \dot{q}_d/\dot{q}_g}{A - \dot{q}_d/\dot{q}_g}$$

L'exécution d'une ligne droite consiste à satisfaire l'équation $\omega = 0$, c'est à dire: $\dot{q}_d/\dot{q}_g = \hat{A}$. Si $\hat{A}$ diffère de A alors la trajectoire exécutée sera un arc de cercle de rayon:

$$\rho = \frac{L}{2}\frac{A + \hat{A}}{A - \hat{A}} \tag{E.4}$$

Insistons sur le fait que du point de vue du robot, la trajectoire exécutée est une droite. La figure E.2 produite par $\mathcal{G}r\mathcal{H}e$ montre la trajectoire calculée par l'odométrie et l'arc de cercle visualisé par le système de localisation externe dans le cas où on a introduit artificiellement un écart $A - \hat{A}$ important.

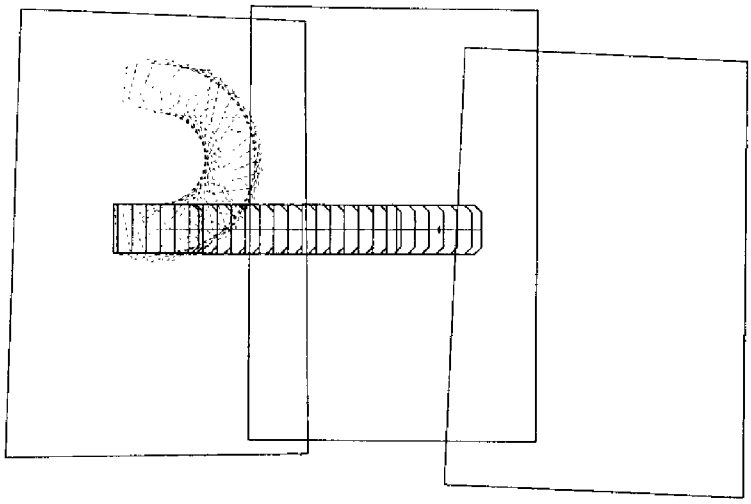


FIG. E.2 – *La trajectoire selon la localisation externe (pointillés) et selon l'odométrie (trait plein).*

A chaque estimée $\hat{A}_i$ correspond un arc de cercle de rayon ρ_i que l'on peut mesurer à l'aide des caméras externes. Notons que le signe de ρ_i dépend de la relation d'ordre entre A et $\hat{A}$. D'après l'équation E.4, pour chaque couple $(\hat{a}_i, \rho_i)$ on peut exprimer la relation suivante entre A et L:

$$f_i(A, L) = (A + \hat{A}_i)(L - 2\rho_i) - 4\rho_i\hat{A}_i = 0$$

$f_i(A, L) = 0$ est une hyperbole centrée sur $(-\hat{A}_i, 2\rho_i)$ dont les asymptotes sont parallèles aux axes principaux. Ces hyperboles s'intersectent toutes au point (A, L). La figure E.3 montre six de ces hyperboles obtenues expérimentalement.

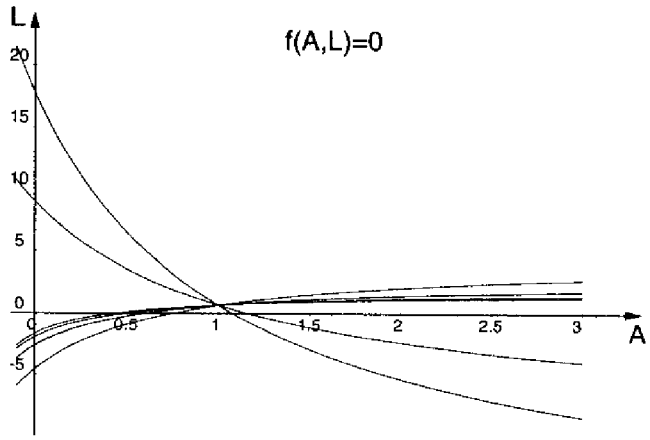


FIG. E.3 – *Les hyperboles obtenues expérimentalement.*

A partir de ce résultat on peut déterminer les valeurs des rayons des roues en exécutant de

vrais lignes droites: notons $\hat{D}$ la longueur de consigne de ce segment (*i.e.* la longueur calculée par l'odométrie) et D la longueur exécutée, nous avons alors les relations suivantes:

$$\begin{cases} \hat{D} = q_d \hat{R}_d = q_g \hat{R}_g \\ D = q_d R_d = q_g R_g \end{cases} \quad \Longrightarrow \quad \begin{cases} R_d = \hat{R}_d D / \hat{D} \\ R_g = \hat{R}_g D / \hat{D} \end{cases}$$

Références bibliographiques

[Aguilar 95] L. Aguilar, R. Alami, S. Fleury, M. Herrb, F. Ingrand & F. Robert. *Ten Autonomous MobileRobot (and even more) in a Route Network Like Environment.* In IEEE International Workshop on Intelligent Robots and Systems (IROS '95), Pittsburg, (USA), August 1995.

[Alami 93] R. Alami, R. Chatila & B. Espiau. *Designing an intellingent control architecture for autonomous robots.* In International Conference on Advance Robotics. ICAR'93, Tokyo (Japon), November 1993.

[Alami 94] R. Alami, L. Aguilar, F. Robert & S. Fleury. *Multi-robot Navigation through Incremental Merging of Motion Plans.* In Workshop on Advanced Automation Forum on Cooperation (ECLA'94), Madrid, Espaa, Nov 28-Dec 2, 1994, 1994.

[Alami 95a] R. Alami, L. Aguilar, H. Bullata, S. Fleury, M. Herrb, F. Ingrand, M. Khatib & F. Robert. *A General framework for multi-robot cooperation and its implementation on a set of three Hilare robots.* In 4th International Symposium on Experimental Robotics (ISER'95), Stanford, California, Junn 30- July 2, 1995, August 1995.

[Alami 95b] R. Alami, S. Fleury, F. Robert & L. Aguilar. *MARTHA Esprit Project 6668: Interactions between the robot supervision and the vehicle sub-systems.* Rapport technique 95290, Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), Toulouse (France), 1995.

[Alami 95c] R. Alami, F. Robert, F. Ingrand & S. Suzuki. *Multi-robot cooperation through incremental plan-merging.* In IEEE Int. Conf. on Robotics and Automation (ICRA'95), Nagoya (Japan), June 1995.

[Alami 95d] R. Alami, F. Robert, F. Ingrand & S. Suzuki. *A paradigm for plan-merging and its use for multi-robot cooperation.* In IEEE Int. Conf. on Systems, Mans and Cybernetics, San Antonio (USA), 1995.

[Albus 87] J. S. Albus, H. G. McCain & R. Lumia. *NASA/NSB Standard Reference Model for Telerobot Control System Architecture (NASREM).* Rapport technique NSB Technical Note 1235, US DEPARTEMENT OF COMMERCE National Bureau of Standards, 1987.

[Albus 95] J. Albus. *RCS: A Reference Model Architecture for Intelligent Systems.* In AAAI Spring Symposium: Lessons Learned from Implemented Software Architectures for Physical Agents, Stanford University, March 1995. ftp: hobbes.jsc.nasa.gov/pub/korten/spring_symposium/submissions.

[Bauzil 88] G. Bauzil, C. Lemaire & G. Vialaret. *Robot mobile HILARE, architecture et fonctionnalités de base.* Rapport Interne 88132, Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), Toulouse (France), 1988.

[Bauzil 92] G. Bauzil, R.Ferraz De Camargo, C.Lemaire & G.Vialaret. *Robot mobile HILARE II. Matériel et bibliothèques associées.* Rapport Interne 92265, Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), Toulouse (France), July 1992.

[Berry 87] G. Berry, P. Couronné & G. Gonthier. *Programmation synchrone des systèmes réactifs: le langage ESTEREL.* Technique et Science Informatiques, vol. 6, no. 4, 1987.

[Betge-Brezetz 94] S. Betge-Brezetz, R. Chatila & M.Devy. *Natural Scene Understanding for Mobile Robot Navigation.* In IEEE International Conference on Robotics and Automation, San Dieg o, California, 1994.

[Betge-Brezetz 95] S. Betge-Brezetz, R. Chatila & M.Devy. *Object-Based Modelling and Localization in Natural Environments.* In IEEE International Conference on Robotics and Automation, Nagoya, Japan, 1995.

[Bouscayrol 95] A. Bouscayrol. *Structures d'alimentation et strategies de commande pour des systemes multimachines asynchrones : application a la motorisation d'un robot mobile.* PhD thesis, These de doctorat en Genie Electrique, LEEI-INP Toulouse, 1995.

[Brooks 85] R.A. Brooks. *A layered intelligent control system for a mobile robot.* In Robotics Research : The Third International Symposium, Gouvieux (France), 1985.

[Brooks 91] R. Brooks. *Intelligence without Reason.* In 12th International Joint Conference on Artificial Intelligence (IJCAI), Sydney (Australia), 1991.

[Bullata 96] H. Bullata & M. Devy. *Incremental Construction of a Landmark-based and Topological Model of Indoor Environments by a Mobile Robot.* In IEEE International Conference on Robotics and Automation, West Lafayette, USA, April 1996.

[Canudas de Wit 91] C. Canudas de Wit & R. Roskam. *Path following of a 2 DOF wheeled mobile robot under path and input torque constraints.* In IEEE International Conference on Robotics and Automation, Sacramento, (USA), 1991.

[Chatila 90] R. Chatila & R. Ferraz De Camargo. *Open Architecture Design and Intertask/Intermodule Communication for an Autonomous Mobile Robot.* In IEEE International Workshop On Intelligent Robots and Systems, Tsuchiura, Japan, July 1990.

[Chatila 92] R. Chatila, R. Alami, B. Degallaix & H. Laruelle. *Integrated Planning and Execution Control of Autonomous Robot Actions.* In IEEE International Conference on Robotics and Automation, Nice, (France), 1992.

[Chatila 93a] R. Chatila. *Representation + Reason + Reaction → Robot Intelligence.* In 6th International Symposium on Robotics Research. ISSR, Hidden Valley (Pennsylvania, USA), October 1993.

[Chatila 93b] R. Chatila, S. Fleury, M. Herrb, S. Lacroix & C. Proust. *Autonomous Navigation in Natural Environment.* In Third International Symposium on Experimental Robotics (ISER'93), Kyoto, Japan, Oct. 28-30, 1993.

[Chochon 83] H. Chochon & B. Leconte. *Etude d'un module de locomotion pour un robot mobile.* Rapport de fin d'étude ENSAE, Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), Toulouse (France), June 1983.

[Coste-Maniere 92] E. Coste-Maniere, B. Espiau & E. Rutten. *A Task-level Robot Programming Language and its Reactive Execution.* In IEEE International Conference on Robotics and Automation, Nice, (France), 1992.

[Dacre-Wright 93] B. Dacre-Wright. *Planification de Trajectoire pour un Robot Mobile sur Terrain Accidenté.* PhD thesis, Thèse de l'Université Paul Sabatier, Toulouse (France), 1993.

[Degallaix 93] B. Degallaix. *Une architecture pour la téléprogrammation au niveau tache d'un robot mobile autonome.* Thèse de l'Université Paul Sabatier, Toulouse (France) 1372, Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), Toulouse (France), 1993.

[Delingette 91] H. Delingette, M. Herbert & K. Ikeuchi. *Trajectory generation with curvature constraint based on energy minimization.* In IEEE International Workshop on Intelligent Robots and Systems (IROS '91), Osaka (Japan), 1991.

[Dzierzgowski 90] D. Dzierzgowski. *Quatre exemples de langages ou environements pour le développement de programmes où le temps intervient.* Technique et Science Informatiques, vol. 4, pages p. 289 312, 1990.

[Elloy 88] J.-P. Elloy. *Le temps réel.* Technique et Science Informatiques, vol. 7, no. 5, pages p. 493–500, 1988. Article collectif: Groupe de Réflexion Temps Réel du CNRS, sous la rédaction de J.-P. Elloy.

[Espiau 95] B. Espiau, K. Kapellos & M. Jourdan. *Formal Verification in Robotics: Why and How?* In 7th International Symposium of Robotics Research (ISRR'95), Munich, Germany, October 1995.

[Faugeras 86] O.D. Faugeras & F. Lustman. *Inferring Planes by Hypothesis Prediction and Verification for a Mobile robot.* In 7th European Conference on Artificial Intelligence (ECAI), Brighton (United Kingdom), July 1986.

[Ferraz De Camargo 91] R. Ferraz De Camargo. *Architecture Matérielle et Logicielle pour le Contrôle de l'Exécution d'un Robot Mobile Autonome.* Thèse de l'Université Paul Sabatier, Toulouse (France), Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), 1991.

[Fikes 72] R.E. Fikes, P. Hart & N.J. Nilsson. *Learning and Executing Generalized Robot Plans.* Artificial Intelligence, vol. 3, no. 4, 1972.

[Fillatreau 93] P. Fillatreau & M. Devy. *Localization of an Autonomous Mobile Robot from 3D Depth Images using heterogeneous Features.* In IEEE International Workshop on Intelligent Robots and Systems (IROS '93), Yokohama, , Japan), July 1993.

[Fleury 92] S. Fleury & T. Baron. *Absolute external mobile robot localization using a single image.* SPIE, Boston (USA), 15-20 Novembre 1992, 15p., 1992.

[Fleury 93] S. Fleury, T. Baron & M. Herrb. *Monocular localization of a mobile robot.* International Conference on Intelligent Autonomous Systems (IAS-3), Pittsburgh (USA), 15-18 Février 1993, pp.470-479, 1993.

[Fleury 94] S. Fleury, M. Herrb & R. Chatila. *Design of a Modular Architecture for Autonomous Robot.* In IEEE International Conference on Robotics and Automation, San Diego California, (USA), 1994.

[Fleury 95a] S. Fleury, M. Herrb & R. Alami. *MARTHA Esprit Project 6668: Preliminary guide to the RCS Integration.* Rapport technique 95293, Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), Toulouse (France), 1995.

[Fleury 95b] S. Fleury, P. Souères, J.-P. Laumond & R. Chatila. *Primitives for Smoothing Mobile Robot Trajectorie.* IEEE Transactions on Robotics and Automation, vol. 11, no. 3, pages p.441–448, June 1995.

[Georgeff 87] M.P. Georgeff & A.L. Lansky. *Procedural Knowledge.* Rapport technique 411, Artificial Intelligence Center SRI International, January 1987.

[Ghallab 88] M. Ghallab, R. Alami & R. Chatila. *Dealing with Time in Planning and Execution Monitoring.* In R. Bolles, editeur, Robotics Research: The Fourth International Symposium. MIT Press, Mass., 1988.

[Giralt 92] G. Giralt & L. Boissier. *THE FRENCH PLANETARY ROVER VAP: Concept and Current Developments.* In IEEE International Workshop on Intelligent Robots and Systems (IROS '92), Raleigh (North Carolina, USA), pages 1391–1398, July 1992.

[Giralt 93] G. Giralt, R. Chatila & R. Alami. *Remote Intervention, Robot Autonomy, And Teleprogramming: Generic Concepts And Real-World Application Cases.* In IEEE International Workshop on Intelligent Robots and Systems (IROS '93), Yokohama, (Japan), pages 314–320, July 1993.

[Guernic 91] P. Le Guernic, T. Gautier, M. Le Borgne & C. Le Maire. *Programming Real-Time Applications with SIGNAL.* Proceedings of the IEEE, vol. 79, no. 9, September 1991.

[Halbwachs 91] N. Halbwachs, P. Caspi, P. Raymond & D. Pilaud. *The Synchronous Data Flow Programming Language LUSTRE.* Proceedings of the IEEE, vol. 79, no. 9, September 1991.

[Harel 87] D. Harel. *Statecharts: a Visual Formalism for Complex Systems.* Journal of Science of Computer Programming, vol. 8, no. 1, pages p. 231–274, 1987.

[Harel 90] D. Harel & *et al. Statemate, a working environment for the development of complex reactive systems.* IEEE transactions on Software Engineering, vol. 16, no. 4, pages p. 403–413, April 1990.

[Hasemann 95] J.-M. Hasemann. *Robot control architectures application requirements, approaches, and technologies.* In XIV Intelligent Robots and Computer Vision: Algorithms, Techniques, Active Vision, Materials Handling (SPIE), Philidelphia Pennsylvania, USA, October 1995.

[Horaud 89] R. Horaud, B. Conio, E. Leboulleux & B. Lacolle. *An Analytic Solution for the Perspective 4-Point Problem.* Computer Vision, Graphics, and Image Processing, vol. 47, pages 33–44, 1989.

[Ingrand 92a] F. Ingrand, M. Georgeff & A. Rao. *An Architecture for Real-Time Reasoning and System Control.* IEEE Expert, Knowledge-Based Diagnosis in Process Engineering, vol. 7, no. 6, pages p. 33-44, December 1992.

[Ingrand 92b] F. Ingrand, M.P. Georgeff & A.S. Rao. *An architecture for real-time reasoning and system control.* IEEE Expert. Intelligent Systems and Their Applications, vol. 7, pages pp.34–44, 1992.

[Jacobs 91] P. Jacobs, A. Rege & J.-P. Laumond. *Non-holonomic motion planning for Hilare-like mobile robot.* In International Syposium on Intelligent Robotics (ISIR), Bangalore, India, January 1991.

[Kaelbling 88] L. Kaelbling & N. Wilson. *Rex Programmer's Manual.* Rapport technique, Artificial Intelligence Center, SRI International, June 1988.

[Kanayama 89] Y. Kanayama & B. Hartman. *Smooth local planning for autonomous vehicles.* In IEEE International Conference on Robotics and Automation, Scottsdale, (USA), 1989.

[Kanayama 91] Y. Kanayama, Y. Kimura, F. Miyazaki & T. Noguchi. *A stable tracking control method for a non-holonomic mobile robot.* In IEEE International Workshop on Intelligent Robots and Systems (IROS '91), Osaka (Japan), 1991.

[Kapellos 94] K. Kapellos. *Environement de programmation des applications robotiques réactives.* PhD thesis, Ecole des Mines de Paris, November 1994.

[Kapellos 95] K. Kapellos, S. Abdou, M. Jourdan & B. Espiau. *Specification, Formal Verification and Implementation of Tasks and Missions for Autonomous Vehicle.* In The Fourth International Symposium on Experimental Robotics , Stanford, California, pages 257–262, June 1995.

[Khatib 86] O. Khatib. *Real time obstacle avoidance for manipulators and mobile robots.* International Journal of Robotics Research, vol. 1, no. 5, 1986.

[Khatib 95] M. Khatib & R. Chatila. *An extended potentiel field approach for mobile robot sensor-based motions.* In Intelligent Autonomous Systems (IAS'4), Karlsruhe (Germany), 1995.

[Khoumsi 88] A. Khoumsi. *Pilotage, asservissement sensoriel et localisation d'un robot mobile autonome.* Thèse de l'Université Paul Sabatier, Toulouse (France), Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), Toulouse (France), June 1988.

[Lacombe 91] J. L. Lacombe & T. Blais. *PROGRAMME AMR DE ROBOTS MOBILES AVANCES. Solutions retenues pour le développement des modules de perception et d'interface homme-machine du démonstrateur ADAM.* In 4th International Symposium on Offshore, Robotics and Artificial Intelligence (ORIA), Marseille (France), pages 223–230. Institut International de Robotique et d'Intelligence Artificielle de Marseille, December 1991.

[Lacroix 94] S. Lacroix, R. Chatila, S. Fleury, M. Herrb & T. Simeon. *Autonomous Navigation in Outdoor Environment: Adaptative Approach and Experiment.* In IEEE International Conference on Robotics and Automation, San Diego, California, 1994.

[Lacroix 95] S. Lacroix. *Statégies de perception et de déplacement pour la navigation d'un robot mobile autonome en environnement naturel.* Thèse de l'Université Paul Sabatier, Toulouse (France), Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), avril 1995.

[Laruelle 94] H. Laruelle. *Planification Temporelle et Exécution de Tâches en Robotique.* PhD thesis, Université Paul Sabatier, Toulouse, April 1994.

[Lasserre 95] P. Lasserre & P. Grandjean. *Stereo Vision Improvements.* In submitted to the '95 International Conference on Advanced Robotics, Barcelona, Spain, 1995.

[Latombe 91] J.-C. Latombe. *A Fast Path Planner for a Car-Like Indoor Mobile Robot.* In AAAI Press, editeur, 9th Natl. Conf. on Artificial Intelligence, 1991.

[Laumond 94] J.-P. Laumond, P. Jacob, M. Taix & R. Murray. *A Motion Planner for Nonholonomic Mobile Robots.* IEEE Transactions on Robotics and Automation, vol. 10, no. 5, October 1994.

[Lavarenne 94] C. Lavarenne & Y. Sorel. *Optimisation et génération d'exécutifs distribués temps-réel pour algorithmes spécifiés avec les langages synchrones.* In Acts Reel-Time Systems, September 1994.

[Maraninchi 90] F. Maraninchi. *Argos: un langage graphique pour la conception, la description et la validation des systèmes réactifs.* PhD thesis, Université Joseph Fourier - Grenoble I, France, January 1990.

[Marty 94] J.-C. Marty. *Utilisation des satecharts pour une spécification structurée du contrôle des cellules flexibles.* PhD thesis, Thèse de l'Université Paul Sabatier, Toulouse (France), Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), Toulouse (France), June 1994.

[Medeiros 93] A. De Medeiros. *Protoytpe d'un module exécutif pour le robot Hilare2.* Rapport de DEA, Université Paul Sabatier, Toulouse, France, 1993.

[Milan 93] C. Milan, G. Richard, M. Paindavoine, Y. Sorel & C. Lavarenne. *Implantation d'algorithmes de traitements d'images sur une architecture multi-DSP avec l'environnement d'aide à l'implantation SynDEx.* In Actes du 15ème GRETSI, Juan-les-Pins, France, Sept., 1993.

[Miller 93] D. Miller. *Intelligent Mobile Robots: Perception of Performance.* In '93 International Conference on Advanced Robotics (ICAR),Tokyo (Japan), 1993.

[Mohr 88] R. Mohr. *Sur l'appariement modèle - perception.* In 2ème Atelier Scientifique : "Traitement d'Images : du Pixel à l'Interprétation" (TIPI), Aussois (France), pages XXIX–1 – XXIX–17. C.N.R.S., April 1988.

[Moutarlier 91] P. Moutarlier. *Modélisation Autonome de l' Environnement par un Robot Mobile.* Thèse de l'Université Paul Sabatier, Toulouse (France) 91381, Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), October 1991.

[Nashashibi 93] F. Nashashibi. *Perception et Modélisation de l'Environnement Tridimensionnel pour la Navigation Autonome d'un Robot Mobile* . Thèse de l'Université Paul Sabatier, Toulouse (France) 93032, Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), February 1993.

[Nelson 89] W. Nelson. *Continuous-Curvatures Paths for Autonomous Vehicules.* In IEEE International Conference on Robotics and Automation, Scottsdale, (USA), pages p. 1260–1264, 1989.

[Noreils 89] F. Noreils. *Contrôle d'Exécution de Plans d'Actions et Architecture pour Robots Mobiles.* Thèse de l'Université Paul Sabatier, Toulouse (France) 560, Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), November 1989.

[Orteu 91] J.J. Orteu & M. Devy. *Application of Computer Vision to Automatic Selective Cutting with a Roadheader in a Potash Mine.* In 5th International Conference on Advanced Robotics (ICAR), Pisa (Italy), June 1991.

[Padiou 90] G. Padiou & A. Sayah. Techniques de synchronisation pour les applications parallèles. Cepadus Editions, Toulouse, France, 1990.

[Perebaskine 92] V. Perebaskine. *Une architecture modulaire pour le contrôle d'un robot mobile autonome.* Thèse de l'Université Paul Sabatier, Toulouse (France), Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), Toulouse (France), 1992.

[Perret 95] J. Perret. *Téléprogrammation au niveau tache pour un robot mobile autonome d'intervention sur site distant.* PhD thesis, Université Paul Sabatier, Toulouse, November 1995.

[Philippe 89] H. Philippe. *Algorithmes pour la compilation de bases de connaissances en logique propositionnelle et du premier ordre: les systèmes KHEOPS et CLOPS.* Thèse de l'Université Paul Sabatier, Toulouse (France), Laboratoire d'Automatique et d'Analyse des Systèmes (C.N.R.S.), 1989.

[Pissard-Gibollet 95] R. Pissard-Gibollet, K. Kapellos, P. Rives & J.-J. Borrely. *Real-time programming of mobile robot actions using advanced control techniques.* In The Fourth International Symposium on Experimental Robotics , Stanford, California, pages 352–357, June 1995.

[Rives 83] P. Rives, P. Bouthemy, B. Prasada & E. Dubois. *Recovering the orientation and the position of a rigid body in space form a single view.* Rapport technique, INRS-Télécommunications, 3, place du Commerce, Ile-des-Soeurs, Verdun, H3E 1H6 Québec , Canada, 1983.

[Robert 96] F. Robert, R. Alami, F. Ingrand & M. Ghallab. *A Multi-robot Cooperative Scheme based on Distributed ans Incremental Plan Merging.* In AIPS'96, 1996. Submited.

[Roux 92] O. Roux, D. Creusot, F. Cassez & J.-P. Elloy. *Le langage réactif asynchrone ELECTRE.* Technique et science informatiques, vol. 11, no. 5, pages p. 35–66, 1992.

[Samson 90] C. Samson & K. Ait-Abderrahim. *Mobile-Robot Control. Part 1 : Feedback Control of a Nonholonomic Wheeled Cart in Cartesian Space.* Rapport de recherche 1288, Institut National de Recherche en Informatique et en Automatique (I.N.R.I.A.), Sophia Antipolis (France), October 1990.

[Samson 91] C. Samson & K. Ait-Abderrahim. *Feedback stabilization of a nonholonomic wheeled mobile robot.* In IEEE International Workshop on Intelligent Robots and Systems (IROS '91), Osaka (Japan), 1991.

[Schneider 95] S. Schneider, V. Chen & G. Pardo-Catellote. *The ControlShell Components-Based Real-Time Programming System.* In IEEE Int. Conf. on Robotics and Automation (ICRA'95), Nagoya (Japan), 1995.

[Segovia 91] A. Segovia, M. Rombaut, A. Preciado & D. Meizel. *Comparative study of the different methods of path generation for a mobile robot in a free environment.* In '91 International Conference on Advanced Robotics (ICAR),Pisa (Italy), 1991.

[Shin 90] D. Shin & S. Singh. *Path generation for robot vehicules using composite clothoid segments.* Rapport technique CMU-RI-TR-90-31, Robotics Institute, Carnegie Mellon University, 1990.

[Siméon 93] T. Siméon & B. Dacre Wright. *A Practical Motion Planner for All-terrain Mobile Robots.* In IEEE International Workshop on Intelligent Robots and Systems (IROS '93) Japan, July 1993.

[Simmons 90] R. Simmons, L. J. Lin & C. Fedor. *Autonomous Task Control for Mobile Robots.* In IEEE International Symposium on Intelligent Control, Philadelphia, PA, 1990.

[Simmons 95] R. Simmons. *Towards Reliable Autonomous Agents.* In AAAI Spring Symposium: Lessons Learned from Implemented Software Architectures for Physical Agents, Stanford University, March 1995. ftp: hobbes.jsc.nasa.gov/pub/korten/spring_symposium/submissions.

[Simon 93] D. Simon, B. Espiau, E. Castillo & K. Kapellos. *Computed-Aided Design of a Generic Robot Controller Handling Reactivity and Real-Time Control Issues.* IEEE Transaction on Control Systems Technology, vol. 1, no. 4, 1993.

[Stewart 95a] D. Stewart & P. Khosla. *The Chimera Methodology: Designing Dynamically Reconfigurable and Reusable Real-Time Software using Port-Based Objects.* to appear in Int. Journal of Software Engineering and Knowledge Engineering (Dec 1995 or May 1996), 1995. http://www.glue.umd.edu/~dstewart/bib.

[Stewart 95b] D. Stewart & P. Khosla. *Rapid Development of Robotic Applications using Component-Based Real-Time Software.* In IEEE International Workshop on Intelligent Robots and Systems (IROS '95), Pittsburg, (USA), August 1995.

[Teel 92] R. Teel, R. Murray & G. Walsh. *Nonholonomic Control Systems: From Steering to Stabilization with Sinusoids.* In 31st Conference on Decision and Control. Tucson, Arizona., pages p. 1603–1609, December 1992.

[Tsai 87] R.Y. Tsai. *A versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-shelf TV Cameras and Lenses.* IEEE Journal of Robotics and Automation, vol. 3, no. 4, pages 323–344, August 1987.

Architecture de contrôle distribuée pour robots mobiles autonomes: principes, conception et applications

Un robot mobile autonome doit réaliser des tâches non répétitives dans un environnement imparfaitement connu et non-coopératif, voire hostile. Dans ce contexte les missions attribuées au robot ne peuvent être définies que de façon abstraite et peu détaillée, et le robot doit être doté de moyens pour les interpréter, appréhender l'environnement, décider des actions adéquates et réagir aux événements asynchrones. Afin de concilier décision et réaction, l'architecture de contrôle proposée, c'est à dire la manière dont sont organisées les composantes logicielles du robot, comporte deux niveaux hiérarchiques: les niveaux décisionnel et fonctionnel. Ce second niveau, objet principal de la thèse, fournit l'ensemble des capacités opératoires du système (perception, modélisation, mouvements et actions).
La première partie du mémoire présente l'architecture globale et fournit un état de l'art et une analyse critique focalisée sur l'organisation des systèmes réactifs. La seconde partie explicite les conditions requises au niveau de la couche fonctionnelle pour satisfaire l'autonomie, la réactivité et la programmabilité du robot. Ces caractéristiques, associées à la grande diversité et aux contraintes temporelles des fonctions opératoires, ont conduit à une structuration en *modules*. La formalisation structurelle, comportementale et fonctionnelle des modules a permis en particulier de concevoir des méthodes générales d'intégration de fonctions. Les fonctions ainsi encapsulées dans les modules composent une ensemble de services homogènes, réactifs et observables à la disposition du niveau décisionnel qui accomplit les tâches du robot en les combinant dynamiquement en un arbre d'activités. Les modules sont décrits et produits au moyen d'un langage de spécification associé à un générateur automatique de modules ($G^{en}{}_{o}M$). La dernière partie présente trois intégrations complètes. La première concerne Hilare, un robot expérimental d'intérieur équipé de nombreux capteurs et fonctionnalités. Des méthodes originales de localisation et de contrôle de déplacement pour véhicule non-holonome sont détaillées. La seconde porte sur la navigation en milieu naturel du robot tout terrain ADAM. La dernière, relative à la coopération multi-robots, a conduit à une simulation réaliste d'une quinzaine de robots (sous UNIX) et à une expérimentation réelle avec trois robots Hilare (sous VxWorks).

Mots clefs: Robot mobile autonome, Systèmes réactifs temps-réel, Architecture de contrôle distribuée, Non-holonomie, Coordination multi-robots.

Control Architecture for Autonomous Mobile Robots: Principles, Design and Integration

An autonomous mobile robot should execute non repetitive tasks in ill known and non cooperative environments. In such contexts, the missions given to the robot cannot be defined precisely and the robot must be endowed with capacities for mission interpretation and scene understanding, to be able to decide appropriate actions and to react to external events. The proposed control architecture, i.e., the organisation of the robot system, consists in two hierarchic levels: the decisional and functional levels. The presented work is mainly related to the second level which provides for the operational capacities of the system (perception, modelling, motion and actions). The first part of the manuscript presents the global architecture and provides a critical analysis of the state of the art focussed on reactive systems. The second part clarifies the requirements for the functional level to provide the robot with autonomy, reactivity and programmability. These features, associated with the diversity and the temporal constraints of the operational functions, lead to a modular structure. From a behavioral, functional and structural formalization of the modules, a generic module has been defined. The functions integrated in modules appear as consistent, reactive and homogeneous services to the decisional level which carried out robot tasks by dynamically connecting these functions within a tree of activities. The modules are described and produced by a specification language associated with an automatic generator of modules named $G^{en}{}_{o}M$. The last part presents three complete integrations. The first one concerns the indoor experimental robot Hilare equipped with numerous sensors and functionalities. Original localization and non-holonomic motion control procedures are detailed. The second one describes the system for outdoor navigation with the all-terrain mobile robot ADAM. The last one deals with multi-robot coordination and has lead to a realistic simulation running fifteen robots (under UNIX) and to a real experimentation with three Hilare robots (under VxWorks).

Key words : Autonomous mobile robot, Real-time reactive systems, Distributed control architecture, Non-holonomy, Multi-robot coordination.