Contrôle d'exécution pour robots mobiles autonomes: architecture, spécification et validation

Adelardo A.D. de Medeiros

▶ **To cite this version:**

Adelardo A.D. de Medeiros. Contrôle d'exécution pour robots mobiles autonomes: architecture, spécification et validation. Automatique / Robotique. Université Paul Sabatier - Toulouse III, 1997. Français. NNT : . tel-00010029

HAL Id: tel-00010029

https://theses.hal.science/tel-00010029v1

Submitted on 2 Sep 2005

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.

N° d'ordre: 2602

Année 1997

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

Adelardo A. D. de MEDEIROS

Ingénieur de l'Université Fédérale du Rio Grande do Norte - Brésil
Mestre par l'Institut Technologique d'Aéronautique - Brésil

Contrôle d'exécution pour robots mobiles autonomes: architecture, spécification et validation

---

Soutenue le 19 février 1997 devant le jury composé de:

Président	**Georges**	**GIRALT**
Directeur de thèse	**Raja**	**CHATILA**
Rapporteurs	**Bernard**	**DUBUISSON**
	Jean-Louis	**LACOMBE**
	Emmanuel	**MAZER**
Examinateurs	**Guy**	**JUANOLE**
	Malik	**GHALLAB**

Rapport LAAS N° 97063
Cette thèse a été préparée au LAAS-CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4

Il y a deux catégories de chercheurs dont les uns ne seraient que des manœuvres, tandis que les autres auraient pour mission d'inventer. L'invention doit être partout, jusque dans les plus humbles recherches de faits, jusque dans l'expérience la plus simple. Là où il n'y a pas un effort personnel et même original, il n'y a même pas un commencement de science.

Henri BERGSON

Avant Propos

Le travail présenté dans ce mémoire a été effectué dans l'équipe Robotique et Intelligence Artificielle du LAAS-CNRS, à Toulouse. À ce sujet, je remercie l'accueil d'Alain Costes, directeur du LAAS, de Georges Giralt, chef du groupe RIA, et de leurs successeurs respectifs, Jean-Claude Laprie et Malik Ghallab. Je tiens aussi à remercier le soutien du gouvernement brésilien, par le biais de l'agence CAPES et de l'Université Fédérale du Rio Grande do Norte, qui a rendu possible ce travail.

Je voudrais exprimer ma gratitude envers mon directeur de thèse, Raja Chatila, dont j'ai pu, au long de ces années, apprécier la compétence, la confiance en moi et la capacité d'ouverture à des nouvelles idées.

J'adresse aussi ma reconnaissance à Bernard Dubuisson, Jean-Louis Lacombe et Emmanuel Mazer, qui ont accepté d'être les rapporteurs de cette thèse, soutenue devant le jury auquel ont aussi participé Georges Giralt, son président, Guy Juanole, Malik Ghallab et Raja Chatila. Qu'ils soient tous assurés de ma gratitude pour avoir jugé et enrichi mon travail.

Je voudrais aussi marquer ma sympathie envers les personnes avec qui j'ai travaillé et discuté au LAAS. Parmi eux, Sara Fleury et Maher Khatib méritent une reconnaissance particulière, en raison de l'aide précieuse qu'ils m'ont offert dans plusieurs domaines. Je donnerais aussi une mention spéciale à Flavien Huynh, Brigitte Lamare et Xavier Fouchet, les courageux lecteurs de la première version de ce mémoire, et à Sílvia Botelho et Guido Lemos, qui m'ont aidé à préparer la soutenance de la thèse.

Et puis, comment ne pas mentionner tous ceux qui ont contribué à créer l'excellente ambiance qui a regné pendant mes quatre ans et demi à Toulouse? Je ne peux pas m'empêcher d'adresser mes salutations fraternelles à tous les fidèles des déjeuners et des pauses café, des voyages, des soirées entre amis, des petits restos, des pots, des fêtes et de toutes les autres occasions pleines de chaleur humaine qu'on a pu vivre ensemble. À quoi bon les citer en particulier? Ils se seront sûrement reconnus...

Je terminerai ces remerciements par un clin d'œil plein de tendresse à mes parents, qui même à des milliers de kilomètres sont toujours à côté de moi, avec leur confiance et leur soutien indéfectible. Muito obrigado, papai. Muito obrigado, mamãe.

Table des matières

Table des matières i

Table des figures v

Liste des tableaux vii

Introduction 1

1 L'architecture de contrôle 5
- 1.1 Les exigences du système . . . 5
- 1.2 Les approches basées sur les systèmes temps réel réactifs . . . 7
- 1.3 Les approches basées sur des paradigmes architecturaux . . . 9
  - 1.3.1 Les architectures hiérarchisées . . . 9
  - 1.3.2 Les architectures comportementales . . . 11
  - 1.3.3 Les architectures centralisées . . . 12
- 1.4 Les architectures hybrides . . . 14
- 1.5 Concepts généraux de l'architecture LAAS . . . 16
  - 1.5.1 Le paradigme superviseur - planificateur . . . 17
  - 1.5.2 Analyse comparative . . . 18
- 1.6 L'architecture LAAS à trois niveaux . . . 20
  - 1.6.1 Le niveau mission . . . 20
  - 1.6.2 Le niveau tâche . . . 21
  - 1.6.3 L'exécutif . . . 22
  - 1.6.4 Les modules . . . 23
  - 1.6.5 La distribution des données . . . 25
  - 1.6.6 Un exemple d'exécution . . . 25
- 1.7 Conclusion . . . 28

2 Les modules 29
- 2.1 Propriétés générales . . . 29
  - 2.1.1 Le flux de contrôle . . . 30
  - 2.1.2 Le flux de données . . . 31

2.2 Architecture d'un module 32
2.3 Un exemple de réseau de modules 35
2.4 Description formelle et génération automatique 36
2.4.1 Un exemple de description formelle 37
2.5 Le protocole de contrôle 39
2.6 Le contrôle d'exécution et les modules 41
2.7 Conclusion 44

3 L'exécutif 45
3.1 Propriétés générales 45
3.2 La communication avec les modules 48
3.3 L'architecture et le fonctionnement 48
3.3.1 La liste de description des services 50
3.3.2 La liste d'activités 51
3.3.3 Le gestionnaire de requêtes 53
3.3.4 Le gestionnaire de répliques 54
3.3.5 Le contrôleur de ressources 54
3.4 Gestion des situations d'erreur 55
3.4.1 Les erreurs signalées par les modules 55
3.4.2 Les erreurs détectées par l'exécutif 56
3.5 Les réactions réflexes 57
3.6 Gestion des conflits 57
3.6.1 La représentation de l'information dans le contrôleur de ressources . 58
3.6.2 La mise en œuvre du contrôleur de ressources 62
3.7 Le système Kheops 63
3.8 Kheops et l'exécutif 66
3.8.1 Le contrôleur de ressources 67
3.8.2 Les gestionnaires de requêtes et de répliques 70
3.9 La communication avec le niveau tâche 71
3.10 Conclusion 72

4 Le niveau tâche 73
4.1 Le système PRS 73
4.2 La mise en œuvre du niveau tâche 76
4.2.1 La communication avec l'exécutif 76
4.2.2 La manipulation des données 77
4.3 Les réseaux de Pétri colorés 78
4.4 PRS et les réseaux de Pétri 80
4.4.1 Le principe de la modélisation 80
4.4.2 La base de données 81
4.4.3 Les variables 86
4.4.4 Les buts 87
4.4.5 Les nœuds 88
4.4.6 Les arcs 89

4.4.7 Le non déterminisme 91
4.4.8 Les KAs 92
4.4.9 L'activation des KAs 93
4.4.10 Les limitations 96
4.5 Vérification du niveau tâche 97
4.6 Conclusion 99

5 Expérimentations **101**
5.1 Description de l'expérimentation 101
5.2 Les modules mis en œuvre 104
5.2.1 La locomotion 104
5.2.2 La planification de trajectoires 106
5.2.3 Le contrôle du bras manipulateur 106
5.2.4 Perception et modélisation de l'environnement 107
5.2.5 L'exploration 109
5.3 L'exécutif 110
5.4 Le niveau tâche 112
5.4.1 Vérification 113
5.5 Un scénario illustratif 116
5.6 Un exemple d'exécution 118
5.7 Conclusion 118

Conclusions et perspectives **121**

A Le système Kheops **123**
A.1 Les composants 123
A.1.1 La base de règles 123
A.1.2 La mémoire de travail 123
A.1.3 La machine d'inférence 124
A.2 La syntaxe 125
A.3 La compilation 126

Bibliographie **129**

Table des figures

0.1 Une mise en œuvre de l'architecture LAAS . . . 2

1.1 Relations entre les éléments dans l'architecture NASREM . . . 10
1.2 L'architecture de subsomption . . . 11
1.3 L'architecture TCA pour le robot Ambler . . . 13
1.4 Paradigme d'organisation du niveau décisionnel . . . 17
1.5 Les architectures hiérarchisées par rapport à l'architecture LAAS . . . 18
1.6 Les architectures comportementales par rapport à l'architecture LAAS . . . 19
1.7 Les architectures centralisées par rapport à l'architecture LAAS . . . 19
1.8 L'architecture LAAS à trois niveaux . . . 20
1.9 Un exemple d'arbre d'activités à l'intérieur du niveau fonctionnel . . . 24
1.10 Un exemple d'exécution de mission . . . 26

2.1 Une relation client/serveur . . . 30
2.2 Le flux de données: les posters . . . 31
2.3 Graphe de contrôle d'une activité . . . 33
2.4 Une instance du graphe de contrôle étendu d'une activité . . . 34
2.5 La famille de robots *Hilare*, avec *Hilare* 2 au centre . . . 35
2.6 Niveau fonctionnel du robot *Hilare* 2 . . . 36
2.7 La structure du générateur de modules . . . 37
2.8 Prévisibilité des fonctions . . . 42
2.9 Persistance des fonctions . . . 42
2.10 Utilisation prévisible et périodique des ressources par les activités filles . . . 43
2.11 Utilisation non persistante d'une ressource par une activité fille . . . 43
2.12 Utilisation non prévisible d'une ressource par les activités fille . . . 43

3.1 L'exécutif vu comme un automate à états finis . . . 47
3.2 L'architecture de l'exécutif . . . 49
3.3 Le graphe de représentation des activités . . . 52
3.4 Une conception du contrôleur de ressources basée sur les ressources . . . 59
3.5 Un exemple de graphe d'incompatibilités entre services . . . 60
3.6 Le graphe d'incompatibilités simplifié . . . 60
3.7 Une transition de l'automate qui représente le contrôleur de ressources . . . 62
3.8 Un exemple de base de règles Kheops et le graphe de décision correspondant 65
3.9 Niveau fonctionnel d'un système générique . . . 66

3.10 Graphe d'incompatibilités pour le système générique 67
3.11 Quelques règles des gestionnaires de requêtes et de répliques 72

4.1 Un exemple de procédure (KA) PRS 75
4.2 Un exemple de KA primitive 76
4.3 Exemple de procédure de communication entre l'exécutif et PRS 77
4.4 Une KA de traitement spécifique d'un service 78
4.5 Un exemple de réseau de Pétri coloré 79
4.6 Tir de transition dans un réseau de Pétri coloré 79
4.7 Deux exemples d'équivalence entre PRS et les RPCs 81
4.8 Caractéristique essentielle des modèles en RPC des arcs PRS 81
4.9 Modélisation d'un prédicat standard dans la base de données 82
4.10 Test d'un prédicat standard 83
4.11 Conclusion d'un prédicat standard dans la base de données 83
4.12 Retrait d'un prédicat standard de la base de données 84
4.13 Modèles des opérations avec les prédicats fermés 84
4.14 Modèles des opérations avec les fait fonctionnels 85
4.15 La représentation des variables dans les jetons 86
4.16 Postage d'un but 87
4.17 Modèle d'un nœud condition 88
4.18 Modèles des nœuds d'ouverture et de fermeture du parallélisme 89
4.19 Modèle d'un arc qui teste une condition complexe 90
4.20 Modèle de l'affectation d'une variable 91
4.21 Modèle d'un arc d'attente 91
4.22 Nœud avec plusieurs arcs partants 92
4.23 Équivalence entre certains champs d'une KA et des arcs supplémentaires . . 93
4.24 Exemple de réseau de Pétri coloré équivalent à une KA 94
4.25 Déclenchement d'une KA activée par un fait 95
4.26 But pour lequel aucune KA n'est déclenchée 95
4.27 But pour lequel une seule KA peut être déclenchée 95
4.28 But pour lequel plusieurs KAs peuvent être déclenchées 96

5.1 Un algorithme parallèle pour l'expérimentation 102
5.2 L'architecture de l'expérimentation 103
5.3 L'organisation mise en place pour l'exécution d'une trajectoire 105
5.4 Capacité de détection d'un objet selon sa position dans l'image 108
5.5 Le graphe d'incompatibilités entre les services 111
5.6 Exemple de règles de l'exécutif 112
5.7 Les procédures PRS pour les tâches de prise d'objet, exploration et modélisation 113
5.8 Représentation des incompatibilités entre services 114
5.9 Réseau de Pétri correspondante à la tâche de modélisation 115
5.10 Un scénario d'exécution de l'expérimentation 117
5.11 Un état du système pendant une expérimentation 119

Liste des tableaux

1.1 Comparaison entre trois types d'architecture de contrôle 14

2.1 Dictionnaire des messages échangées entre clients et serveurs 40

3.1 Les services du système générique 67

5.1 Les requêtes du module PILO 104
5.2 Les requêtes du module PLAN 106
5.3 Les requêtes du module ARM 107
5.4 Les requêtes du module VISUAL 107
5.5 Les requêtes du module EXPLOR 109
5.6 Les services offerts par l'exécutif pour l'expérimentation 110
5.7 Les groupes de services 111

A.1 Quelques exemples d'utilisation d'ensembles en Kheops 126

Introduction

Dans ce travail, nous nous intéressons aux robots mobiles autonomes, capables d'effectuer des tâches non répétitives dans des environnements dynamiques et imparfaitement connus. Dans ce contexte, les missions attribuées au robot doivent pouvoir être définies d'une façon abstraite et peu détaillée, le robot étant lui même capable de les interpréter et de les détailler en fonction du contexte réel d'exécution et de les adapter (voire les annuler) selon sa perception de l'environnement qui l'entoure.

Un robot mobile autonome doit donc gérer de façon intelligente l'utilisation de ses ressources physiques et logicielles: décider quand planifier et quand agir, comment traiter des buts conflictuels, etc. Au fur et à mesure que les tâches et les environnements deviennent complexes, il s'avère nécessaire d'imposer une structure d'organisation au contrôle de la planification, de la perception et de l'action, pour améliorer l'intelligibilité du système et pour avoir plus de garanties d'accomplissement des tâches. L'organisation de ces différents sous-systèmes et du flux de contrôle et de données entre eux constitue l'architecture de contrôle du robot.

Certains auteurs, comme Thorpe et Hebert [Thorpe 95], défendent que, au lieu d'avoir une architecture qui doit guider la conception du robot, il vaut mieux prévoir un ensemble d'objets, chacun fournissant certaines capacités, et laisser le concepteur les assembler selon les besoins pour une application quelconque. L'architecture émergera de l'assemblage des objets, au lieu d'être le point de départ du système entier. Cette façon de travailler, si elle peut donner de bons résultats pour le développement de robots conçus pour une tâche bien précise, ne se prête guère au cas des robots autonomes, où les tâches à accomplir sont multiples et, principalement pour les systèmes expérimentaux, où les capacités du robot sont en constante évolution.

Si la définition préalable d'une architecture de contrôle est donc souhaitable pour un robot mobile autonome, il faut que la conception adoptée soit adaptée aux situations auxquelles le robot est confronté. Il est impossible de prévoir parfaitement et de façon exhaustive l'évolution de l'environnement et même le comportement du robot dans des situations réelles, ce qui oblige le système à être réactif: il doit surveiller l'environnement et le robot lui-même pour détecter en temps réel les événements qui peuvent influencer le fonctionnement du système. L'architecture doit aussi prévoir des mécanismes de contrôle d'exécution des activités qui sont déclenchées dans le système.

Ce travail s'intéresse aux problèmes de contrôle d'exécution dans un type particulier d'architecture de contrôle, celle développée au LAAS. Nous essayerons de montrer comment le contrôle d'exécution se fait présent dans les divers niveaux de l'architecture et leur

façon d'interagir de façon à garantir la réactivité du robot sans qu'il perde ses capacités de raisonnement et de prédiction. Une attention particulière a été donnée aux possibilités de vérification formelle (soit des propriétés logiques, soit des caractéristiques temporelles) des sous-systèmes de l'architecture plus directement impliqués dans le contrôle de l'exécution.

Le mémoire commence (chapitre 1) par récapituler les principales propositions d'architectures de contrôle et les paradigmes qui leur servent de base. On fait une analyse comparative de leur adéquation dans le cas des robots mobiles autonomes et on présente les caractéristiques de l'architecture qui a été développée au LAAS [Alami 93].

Pour combiner les capacités de délibération et réactivité [Chatila 95], l'architecture LAAS décompose le système en deux niveaux principaux:

- le niveau fonctionnel, formé par un ensemble de *modules* et un *exécutif*, qui regroupe les capacités de perception et d'action du robot et ses réactions réflexes (non réfléchies); et
- le niveau décisionnel, éventuellement divisé en sous-niveaux, où sont concentrées les prises de décision et la planification du système. Chaque sous-niveau possède des capacités d'anticipation (un planificateur) et de réaction a des événements (un superviseur) adaptées au type de données qu'il manipule.

La mise en œuvre de l'architecture LAAS la plus générique, et qui a été utilisée pendant nos travaux, est représentée dans la figure 0.1. Le niveaux décisionnel y est décomposé en deux sous-niveaux: niveau tâche et niveau mission. Dans le niveau tâche l'aspect supervision est plus présent que la planification, ce qui le fait être couramment dénommé *superviseur*. Le niveau mission, où l'inverse se vérifie, est normalement appelé *planificateur*, et n'est pas traité d'une façon approfondie dans ce mémoire.

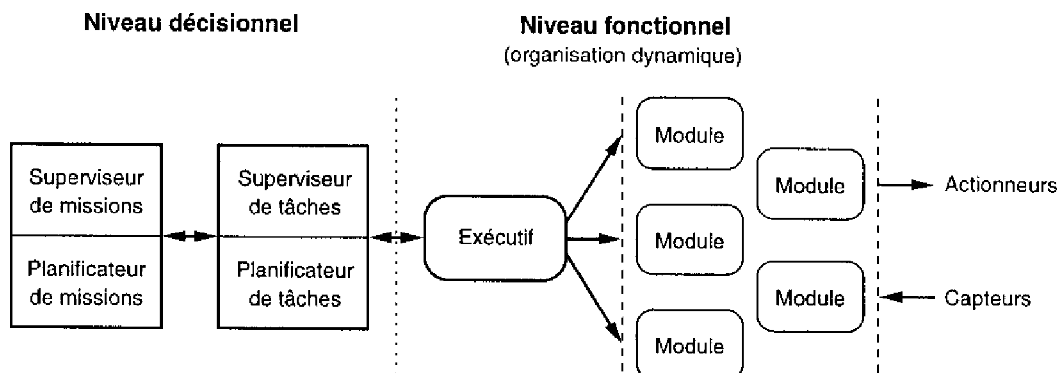


FIG. 0.1 – *Une mise en œuvre de l'architecture LAAS*

Le chapitre 2 présente les travaux qui ont été effectués à propos des modules pour l'architecture LAAS et présente la manière de les intégrer dans le système pour mettre en œuvre la stratégie globale de contrôle d'exécution.

L'exécutif, où se concentre une partie importante de ce travail, est le sujet du chapitre 3. On présente en détail son organisation interne, son fonctionnement et ses relations avec

les modules et le niveau tâche. À cause de sa position dans l'architecture de contrôle du robot, on exige de l'exécutif certaines propriétés logiques (prévisibilité du comportement) et temporelles (temps de réponse borné): on montre comment la mise en œuvre adoptée, principalement grâce à l'utilisation du système Kheops [Ghallab 88], permet de respecter et de prouver partiellement ces propriétés.

Le chapitre 4 traite du niveau tâche. Ce niveau de l'architecture de contrôle, dans les réalisations actuelles, est fortement lié à C-PRS [Ingrand 94], le système de raisonnement procédural utilisé pour sa mise en œuvre. Dans ce mémoire, on présente surtout ce qui concerne les relations entre le niveau tâche et le niveau fonctionnel, comme le flux de données et de contrôle entre eux et l'intégration entre les stratégies de contrôle d'exécution mises en place par l'exécutif et la politique de traitement des événements de PRS. L'autre volet de ce chapitre est la vérification formelle des procédures du superviseur: la versatilité de la syntaxe PRS rendant difficile la détermination des propriétés des procédures, on a voulu expliciter un sous-ensemble de PRS pour lequel on puisse démontrer formellement certaines propriétés, utilisant comme outil les réseaux de Pétri.

Finalement, on a voulu vérifier les performances du système dans des conditions de fonctionnement en temps réel: le chapitre 5 contient quelques résultats de ces expérimentations. Pour conclure, on présente une analyse des résultats obtenus et des perspectives pour les travaux futurs qui pourront les faire évoluer.

Chapitre 1

L'architecture de contrôle

Le contrôle d'un robot mobile autonome implique certains problèmes qui typiquement ne sont pas rencontrés dans d'autres domaines de la recherche en robotique. Par exemple, la nature dynamique du monde, que ce soit dans des environnements intérieurs ou extérieurs, requiert un système de contrôle évolué en temps réel. De plus, pour une navigation autonome, la prise de décisions en temps réel doit être plutôt basée plus sur les informations sensorielles continues que sur la planification hors ligne. Finalement, comme le véhicule explore continuellement différentes régions, il est confronté à une large variété de situations, ce qui requiert une structure de contrôle générale et adaptable.

1.1 Les exigences du système

Le cadre choisi pour le système de contrôle d'un robot mobile autonome doit satisfaire quelques spécifications de comportement, qui ne sont que partiellement identifiées dans la majorité des projets de recherche en robotique temps réel:

Réactivité: À cause la nature non structurée des environnements dans le monde réel, peu d'hypothèses peuvent être faites à propos de la dynamique du monde. Donc, le véhicule doit être réactif à des changements soudains dans l'environnement, avec des bornes temporelles compatibles avec une exécution correcte et sûre de sa tâche.

Robustesse: La robustesse d'un système est perçue comme sa capacité à traiter des entrées imparfaites, des événements non attendus, des incertitudes et des fonctionnements défectueux soudains. Cela suppose une capacité à tirer profit des redondances du système.

Comportement intelligent: Les réactions du robot doivent être guidées par les objectifs de sa tâche. Cela requiert qu'on fasse plusieurs compromis, basés sur les règles du sens commun, de façon à exhiber un comportement intelligent.

Intégration d'informations multiples: Les limitations en précision, sûreté et applicabilité des capteurs individuels doivent être compensées par l'intégration de plusieurs capteurs complémentaires.

Résolution de buts multiples: Dans le cas des robots mobiles, des situations qui requièrent des actions concurrentes et éventuellement conflictuelles sont inévitables.

Le système de contrôle doit fournir des moyens d'accomplir ces buts multiples.

Sûreté: La sûreté d'un système est mesurée par sa capacité à fonctionner sans défaillance ou dégradation de performance pendant une certaine période de temps. Le système de contrôle du robot doit maintenir un niveau de compétence constant indépendamment des particularités de chaque situation.

Ces spécifications de comportement créent quelques exigences au niveau de la conception du système:

Programmabilité: Un robot autonome utile ne doit pas être conçu pour une tâche précise. Il doit être capable d'accomplir plusieurs tâches décrites avec un certain niveau d'abstraction.

Modularité: Comme une exigence générale pour les systèmes complexes, le système de contrôle des robots autonomes doit être divisée en sous-systèmes plus petits qui peuvent être conçus, mis en œuvre et mis au point séparément. Cette conception incrémentale est cruciale pour la maintenance, la détection et corrections des défaillances et pour permettre au système d'être robuste et réactif.

Flexibilité: La robotique expérimentale requiert des changements constants dans le projet pendant la phase de mise en œuvre. Donc, des structures de contrôle flexibles sont nécessaires pour permettre au projet d'être guidé par le succès ou l'échec des éléments individuels.

Évolutivité: Comme une grande période de temps est requise pour concevoir, construire et tester indépendemment les composant individuels d'un robot autonome, un système extensible est souhaitable pour qu'on puisse le construire incrémentalement et y intégrer facilement des nouvelles fonctionnalités.

Autonomie et adaptabilité: Le robot est conçu pour accomplir ses tâches de façon autonome en dépit d'éventuels changements dans les circonstances externes. Cela exige la capacité de raffiner la description d'une tâche pour l'adapter aux conditions d'exécution. En plus, il y a souvent une stratégie de contrôle associée à chaque aptitude du robot; comme l'état du monde change rapidement et d'une façon imprévisible, le système de contrôle doit s'adapter pour commuter rapidement entre ces différentes stratégies de contrôle selon la situation courante.

Comme un robot mobile autonome est un système complexe, la satisfaction simultanée de toutes ces exigences n'est pas évidente, ce qui a conduit à l'apparition de plusieurs propositions de systèmes de contrôle dans la littérature. Normalement, chaque approche se focalise plus spécifiquement sur un des différentes aspects d'un système robotique, comme la vérification des contraintes temporelles où l'organisation du raisonnement intelligent, sans évidemment négliger les autres. Entre autres, on peut distinguer, par la quantité et la qualité des travaux qu'ils ont généré, deux grands axes de recherche:

Les systèmes temps réel réactifs – on appellera *réactif* un système qui maintien une interaction continue avec son environnement, et *temps réel* un système réactif qui doit en plus obéir à des restrictions temporelles [Benveniste 91]. Les robots sont des exemples parfaits de systèmes temps réel réactifs.

Les paradigmes architecturaux – Les robots sont composés de fonctionnalités multiples, qui doivent interagir et communiquer constamment les unes avec les autres. L'organisation correcte de ces éléments requiert la définition d'une architecture pour le système de contrôle, ce qui a été fait selon différentes approches pour les robots autonomes.

Ces deux approches ne sont évidemment pas exclusives et presque tous les systèmes combinent des aspects de ces deux façons de concevoir les systèmes robotiques. L'organisation d'un système temps réel réactif complexe exige aussi la définition d'une architecture, et même la meilleure définition architecturale pour un robot mobile autonome ne sert à rien si le système n'est pas réactif en temps réel. La classification se base surtout sur l'aspect qu'on met en avant dans le projet: soit on part des méthodes et outils des systèmes temps réel réactifs pour les arranger dans une architecture convenable, soit on a un paradigme de définition de l'architecture choisi au préalable qu'on matérialise de façon à garantir la réactivité.

Pour illustrer, même sous le risque de faire des généralisations hâtives, on peut dire qu'en robotique de service ou d'intervention, où prédomine l'analyse de tâches de haut niveau en vue de leur accomplissement dans un monde incertain, est privilégié le paradigme architectural, alors qu'en robotique de manipulation, pour laquelle la dynamique des asservissements et l'interception d'événements asynchrones sont centrales, les recherches se focalisent plus particulièrement sur la spécification et l'implantation de systèmes temps réel réactifs.

1.2 Les approches basées sur les systèmes temps réel réactifs

Le système de contrôle d'un robot doit maintenir des relations continuelles avec des agents (internes au robot ou extérieures) qui progressent indépendamment de lui, avec des temps de réponse imposés par les différents agents. Il peut alors être caractérisé comme un système temps réel réactif, ou STRR. Les méthodes et techniques conçues pour ce genre de système peuvent donc être adaptés aux particularités des systèmes robotiques.

Dans cette optique, plusieurs systèmes de contrôle de robots ont été mis en œuvre avec des langages de programmation temps réel, qui proposent un style de programmation plus adéquat et des outils de vérification du système programmé. On peut citer Esterel [Boussinot 91] et Rex [Kaelbling 88] comme des exemples de langages synchrones qui ont été employés dans le cadre d'applications en robotique. Une autre ligne de recherche a essayé de proposer des méthodes globales de conception et d'intégration de systèmes structurés, basées sur des outils évolués. Deux exemples connus sont Chimera [Stewart 95] et ControlShell [Schneider 95], tous les deux permettant l'intégration modulaire de systèmes temps réel. Orccad [Simon 93] est également adapté à la conception et à l'intégration de systèmes robotiques, mais avec une approche beaucoup plus formelle, fondée sur le langage Esterel, qui permet de procéder à certaines vérifications de propriétés temporelles et comportementales.

Une étude des outils et techniques existantes pour les STRR et de leurs applications en robotique dépasse le cadre de ce travail: on peut voir Benveniste et Berry [Benveniste 91]

pour une introduction à l'approche synchrone pour ces systèmes et la thèse de Sara Fleury [Fleury 96] pour situer ces travaux dans le contexte de la robotique. Nous allons simplement retenir quelques points forts et d'autres moins bons de cette approche:

- La plupart des logiciels pour les systèmes avec le niveau de complexité d'un robot mobile autonome sont développés spécifiquement pour une application. Comme il n'y a pas de standardisation d'interfaces entre les divers composants, l'évolutivité et la flexibilité du produit obtenu, et aussi sa portabilité, sont réduites. La structuration et le formalisme offerts par certains outils et méthodes des STRR peuvent alors faciliter la maintenance et l'évolution du système de contrôle.
- Les possibilités de vérification formelle offertes par certaines approches sont très intéressantes, principalement pour la sécurité des robots impliqués dans des missions coûteuses dans des environnements hostiles (par exemple, exploration planétaire ou sous-marine).
- On peut difficilement, selon notre point de vue, décrire et concevoir un robot (avec le degré d'autonomie qu'on envisage) comme un système purement et entièrement réactif: cela revient à négliger les possibilités de comportement intelligent offertes par la planification et le raisonnement par anticipation.
- Normalement ces méthodes et outils ont des domaines d'applicabilité bien définis, à cause des hypothèses restrictives nécessaires pour adapter les STRR à un certain formalisme logique ou mathématique. L'approche synchrone, par exemple, présuppose que les actions du système sont instantanées (ou au moins indivisibles et assez rapides par rapport à la dynamique du système). Ces conditions peuvent être satisfaites par des composants du robot mettant en œuvre des traitements périodiques de durée constante ou des réactions rapides à des événements asynchrones, par exemple, mais pas par tous les éléments de l'architecture de contrôle.
- Plusieurs approches s'éloignent du cadre qu'on a voulu traiter. 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 la mise en œuvre d'un système multi-fonctionnel dont la logique de contrôle est partiellement établie *en ligne* selon la tâche à accomplir et les données produites. Dans ce contexte, une modélisation exhaustive de toutes les combinaisons d'exécution paraît irréaliste, d'autant que le système est soumis à un flux asynchrone d'événements qui lui parviennent de l'environnement, dont les instants d'occurrence sont peu prévisibles.

À notre avis, les approches basées sur les STRR, même si elles ne sont pas suffisantes pour donner le degré d'autonomie et programmabilité qu'on souhaite pour les robots, ne doivent pas être considérées comme opposées ou concurrentes des approches qui mettent en avant un paradigme architectural, mais plutôt comme des outils importants pour la conception et le développement de certains composants d'une architecture plus globale. À titre d'exemple, plusieurs sous-systèmes du robot peuvent être traités selon le paradigme synchrone si l'on fait quelques adaptations: une possibilité consiste à traiter extérieurement les actions et ne considérer au niveau du contrôle synchrone que les transitions marquant les étapes clefs de ces actions (activation, début et fin d'exécution, interruption, etc.).

Dans l'architecture sur laquelle se fonde notre travail, comme on le verra par la suite, on a voulu incorporer quelques bonnes propriétés que l'on retrouve dans certaines approches pour les STRR, principalement en ce qui concerne l'aide au développement structuré et la vérification:

- l'aide à la conception structurée à été prévue par la génération automatique des modules faite par G^{en}oM [Fleury 96]; et
- les possibilités de vérification formelle ont été considérées par la mise en œuvre synchrone de l'exécutif [Medeiros 96a, Medeiros 96b], faite partiellement avec le système Kheops [Ghallab 88], et par les travaux concernant l'équivalence entre un sous-ensemble de PRS, utilisé pour la mise en œuvre du niveau tâche [Ingrand 96], et les réseaux de Pétri.

1.3 Les approches basées sur des paradigmes architecturaux

Par opposition à l'approche basée sur les systèmes temps réel réactifs, le paradigme "percevoir–modéliser–planifier–agir" décrit sommairement une grande majorité des architectures de contrôle pour robots, même si l'étendue de chacune de ces activités peut changer énormément. Par exemple, la quantité de délibération (modélisation et planification) peut être très importante ou presque négligeable.

Le paradigme "percevoir–modéliser–planifier–agir" présuppose un monde quasi statique ou au moins prévisible pendant la perception et l'action. Comme il s'agit d'une hypothèse presque jamais valable pour le monde réel, dans le contexte de la robotique autonome plusieurs approches ont été essayées pour augmenter la réactivité du système, en brisant le seul canal d'exécution de la méthode originale en divers canaux en parallèle et/ou entrelacés. Entre ces propositions, on peut distinguer trois groupes principaux d'architectures de contrôle: les architectures hiérarchisées, les architectures comportementales (*behavioural*) et les architectures centralisées. Un quatrième groupe, qu'on regroupe sous la dénomination globale d'architectures hybrides, englobe plusieurs systèmes avec des caractéristiques mixtes.

1.3.1 Les architectures hiérarchisées

Les architectures hiérarchisées sont basées sur l'hypothèse que la dynamique du monde diminue avec le niveau d'abstraction. Le paradigme de la décomposition hiérarchique implique la subséquente décomposition des tâches en sous-tâches d'un niveau plus bas et l'organisation du contrôle en niveaux progressifs d'abstraction des données. Comme les processus qui correspondent aux fonctions des divers niveaux manipulent des données avec des contenus d'information différents, ces systèmes peuvent être vus comme une séquence de transformations de données.

Chaque composant du système de contrôle dialogue avec ses voisins (inférieur et supérieur du point de vue hiérarchique). Le flux de contrôle (initiation et terminaison des sous-tâches) est descendant (des niveaux supérieurs vers les actionneurs), alors que le flux de données (résultats d'exécution et lectures des capteurs) est ascendant (avec les capteurs

comme origine). Les niveaux où il n'y a pas beaucoup d'abstraction, comme les asservissements, sont reconnus par leur caractère très immédiat et peu dépendant d'informations extérieures. Les niveaux supérieurs (assimilation et fusion des données des capteurs, modélisation du monde, etc.), par contre, peuvent dépendre de données très symboliques. Le raisonnement dans les niveaux supérieurs produit des plans et des allocations de ressources de longue durée.

Divers travaux, parmi lesquels se détachent ceux d'Albus [Albus 91], soutiennent le paradigme de la décomposition hiérarchique, l'architecture NASREM [Albus 89] étant son exemple le plus cité. NASREM adopte une hiérarchie stricte pour la décomposition des tâches, la perception et la modélisation du monde (fig. 1.1). Cette hiérarchie est basée sur des caractéristiques temporelles — chaque niveau traite des événements qui typiquement se produisent un ordre de grandeur plus lentement que ceux du niveau inférieur.

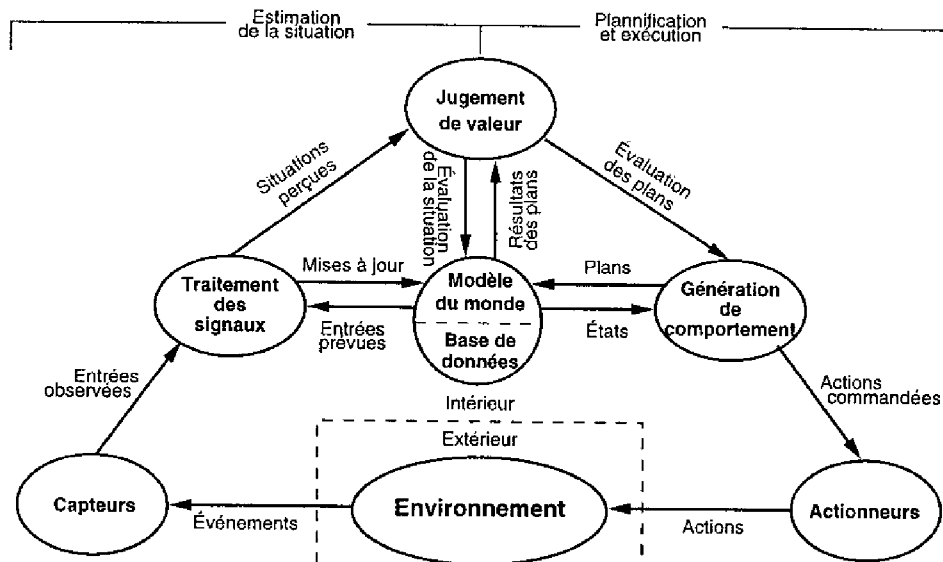


FIG. 1.1 – *Relations entre les éléments dans l'architecture NASREM [Albus 91]*

La décomposition hiérarchique peut gérer des tâches complexes en les divisant en sous-problèmes plus maniables. La planification et la recherche aident le robot à éviter les pièges locaux, ce qui augmente la sûreté du système. Du point de vue de la perception de l'environnement, elle est particulièrement attrayante à cause des possibilités d'abstraction, assimilation, fusion et masquage de morceaux d'information. D'un autre côté, les inconvénients de l'approche purement hiérarchisée sont liés aux aspects suivants:

- La séparation et le long chemin intermédiaire entre l'acquisition de l'information et son usage, avec la conséquente perte d'efficacité et de réactivité.
- Comme le système est partagé en modules, un ensemble d'interfaces doit être défini pour établir le parcours de l'information. La spécification de tout module, cependant, dépend fortement de ses entrées et sorties, de sorte que les modules ne peuvent être

conçus qu'après une spécification stable des interfaces. Cette définition prématurée des interfaces restreint le développement ultérieur des modules, décourageant les améliorations du module qui ne sont pas conformes aux spécifications.

1.3.2 Les architectures comportementales

Une caractéristique distinctive des architectures comportementales est l'absence, au moins de façon centralisée, d'un raisonnement prédictif et d'un modèle global du monde. L'idée est de diviser le système en une collection de comportements simples qui peuvent être exhibés. Le comportement global du robot, qui doit être dirigé vers la tâche à accomplir, n'est pas explicitement planifié: il émerge de l'interaction compétitive entre ses composants.

Chaque comportement détecte directement l'environnement [1], possède sa propre fonction de contrôle et peut décider des actions à prendre, indépendemment de toute décision de n'importe quel autre module: comme les différents comportements peuvent commander des actions différentes, ces systèmes requièrent des mécanismes d'arbitrage.

Les travaux le plus notables dans le domaine des architectures comportementales ont été faits par Brooks [Brooks 91c, Brooks 91b, Brooks 91a], principalement avec son architecture de subsomption (*subsumption architecture*) [Brooks 86], représentée dans la figure 1.2. Dans cette proposition, chaque niveau est composé d'un réseau fixe de modules, qui sont des machines à états finis. L'interaction entre les niveaux se fait toujours du haut vers le bas: un niveau plus haut placé dans la hiérarchie peut subsumer l'entrée ou inhiber la sortie d'un niveau inférieur, qui ignore l'existence des niveaux qui lui sont supérieurs. Pour illustrer, Brooks [Brooks 86], dans son travail pionnier, suggère une instance de son architecture avec un niveau 0 qui garantit que le robot ne rentre pas en contact avec d'autres objets, un niveau 1 qui, combiné avec le niveau 0, donne au robot la capacité d'errer sans but sans percuter des obstacles et un niveau 2 qui ajoute un mode exploratoire au comportement du robot. Les niveaux supérieurs ajoutent de nouvelles capacités au système.

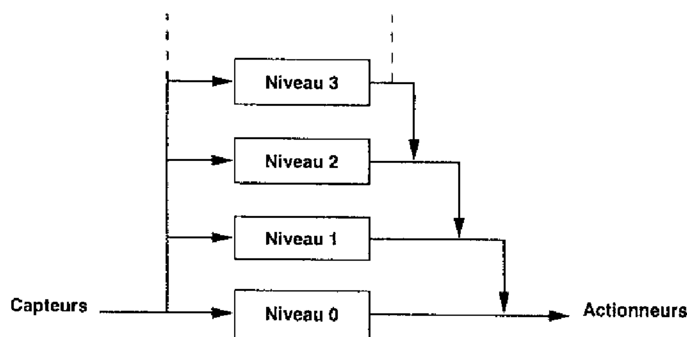


FIG. 1.2 – *L'architecture de subsomption*

1. Un peu dans le même esprit des systèmes réactifs temps réel.

Cette approche a connu un grand succès parce que plusieurs tâches complexes peuvent être mises en œuvre avec une collection de comportements simples qui interagissent. Chaque comportement peut tourner en parallèle avec les autres et extraire de l'environnement seulement les informations qui sont nécessaires à l'accomplissement de sa tâche. Les machines construites selon ce paradigme exhibent un comportement "intelligent" dans certains domaines, avec des exigences en puissance de calcul et des difficultés de développement inférieures à d'autres systèmes. Ces caractéristiques prometteuses ont fait que plusieurs architectures plus ou moins "behaviouristes" aient été proposées ces derniers temps [Badreddin 91, Lewis 93]. Cette approche, cependant, a quelques problèmes intrinsèques:

- Si le système est très évolutif par construction (chaque niveau est simplement ajouté par dessus les autres), il n'est pas flexible: tout nouveau composant doit prendre en compte l'organisation interne des niveaux inférieurs pour pouvoir subsumer ses entrées et sorties, ce qui fait qu'un changement dans le niveau le plus bas risque d'avoir des conséquences sur tout le système.
- L'absence de raisonnement par anticipation oblige toute la capacité de planification du robot à être implicitement et définitivement gravée dans le système de contrôle pendant la phase de conception, ce qui réduit sa programmabilité.
- Les comportements sont très efficaces quand l'information dont ils ont besoin est localement et facilement disponible à partir des capteurs. L'absence de représentations abstraites centralisées, cependant, fait qu'ils ne se prêtent pas très bien à traiter des situations plus complexes, même avec l'utilisation de "capteurs virtuels".
- Quand la complexité augmente, les interactions entre les comportements augmentent aussi, jusqu'au point où il devient difficile de prédire le comportement général du système.
- Un agent global et centralisé de prise de décisions de haut niveau, responsable de la compréhension du système total, est désirable pour prendre en compte les erreurs introduites par fausse interprétation des données sensorielles et pour une évaluation globale de la situation.

1.3.3 Les architectures centralisées

Plusieurs architectures centralisées adoptent le paradigme du tableau noir (*blackboard*). Dans ces architectures il n'y a pas de hiérarchie entre les modules: elles utilisent un tableau central pour coordonner l'interaction entre les divers agents indépendants et les guider vers une solution globale. Cela permet une exécution en parallèle par les composants individuels et aussi un raisonnement et une prise de décision séquentielle. Ce genre d'organisation, qui n'est pas exclusif des systèmes robotiques, se fonde sur trois postulats [Hayes-Roth 85]:

1. Tous les éléments de la solution générés pendant la résolution d'un problème sont enregistrés dans une base de données globale et structurée, appelé le tableau noir.
2. Les éléments de la solution sont générés et enregistrés dans le tableau noir par des processus indépendants, appelés les sources de connaissance.

3. Pour chaque cycle de résolution d'un problème, un mécanisme d'arbitrage choisit une source de connaissance, parmi celles qui peuvent être déclenchées, pour exécuter son action.

Une conception différente a été adoptée par Simmons [Simmons 94] dans son architecture TCA (*Task Control Architecture*), représentée dans la figure 1.3 avec sa configuration pour le robot à six pattes Ambler [Simmons 92]. Dans ces systèmes, le contrôle est centralisé mais les données nécessaires à la résolution des problèmes sont distribués entre les processus. Chaque agent peut contenir un processus de perception, d'action et/ou de prise partielle de décision, pour tout ce qui peut être traité localement et ne dépend pas des autres agents. Les modules communiquent par envoi de messages, qui passent par le module central de contrôle. Les messages peuvent être des données transmises ou des requêtes d'exécution, que le module central renvoie vers le module concerné. Une solution semblable a été adoptée dans l'architecture DPS (*Distributed Problem Solving*) [Baroni 95], même si dans ce travail le module central n'apparaît pas explicitement (il est formé par un ensemble de contrôleurs locaux, qui dialoguent entre eux).

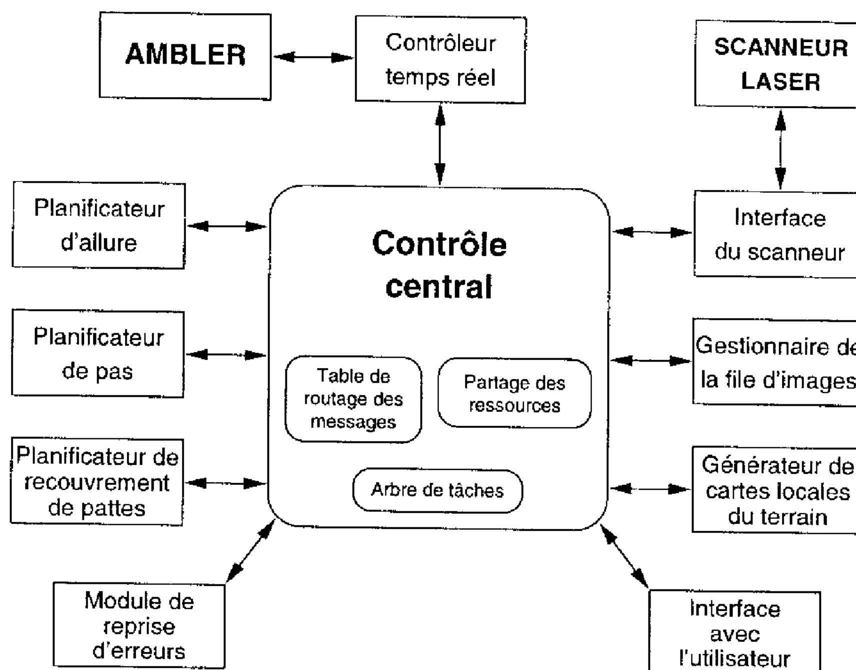


FIG. 1.3 – *L'architecture TCA pour le robot Ambler*

Les architectures centralisées sont très évolutives et flexibles à cause de leur organisation non hiérarchisée [Tigli 93]. L'existence d'un module central, avec une syntaxe bien définie, fournit une orientation sur la conception et combinaison des unités délibératives et réactives et permet une grande adaptabilité du système. Il y a cependant certains domaines

dans lesquels cette approche laisse un peu à désirer:

- Une des faiblesses des architectures centralisées est la surcharge qu'elles imposent au système: avoir plusieurs agents qui fonctionnent en parallèle pour choisir le plus adapté à la situation a un coût en termes de calcul et de stockage de données. Si l'on ne centralise pas les informations, la surcharge se manifeste en termes de vitesse de transfert de données. Si l'on élimine le parallélisme d'exécution, on perd beaucoup d'adaptabilité (et du comportement "intelligent") du robot. Pour que le module central ne devienne pas le goulet d'étranglement du système, il faut prévoir une puissance informatique supérieure à celle qui serait nécessaire avec une autre stratégie.
- L'inclusion d'un planificateur de haut niveau pose des problèmes conceptuels dans ce genre d'architecture: lui, qui devrait contrôler le comportement du robot entier, devient un module de plus sous le contrôle du module central.

1.4 Les architectures hybrides

Les architectures qui adoptent de façon stricte un des paradigmes de contrôle présentés dans la section 1.3 peuvent être analysées selon leur adéquation plus ou moins facile aux critères auxquels les systèmes de contrôle doivent obéir. Les résultats d'une comparaison entre elles, faite par Fayek, Liscano et Karam [Fayek 93], sont présentés dans le tableau 1.1. On peut constater que l'adoption d'un seul des paradigmes n'est pas suffisante pour résoudre le problème du contrôle d'un robot autonome. Plusieurs auteurs [Hasemann 95, Thorpe 95] semblent être d'accord sur le fait que les meilleurs résultats peuvent être obtenus avec des architectures hybrides, qui essaient de prendre le meilleur de chacune des propositions principales.

TAB. 1.1 – *Comparaison entre trois types d'architecture de contrôle d'après [Fayek 93]*

	Architectures Hiérarchisées	**Architectures Comportementales**	**Architectures Centralisées**
Réactivité	Basse	Haute	Moyenne
Intelligence	Haute	Minimale ou aucune	Haute
Capteurs multiples	Oui, avec difficultés	Oui	Oui
Buts multiples	Oui	Oui, avec difficultés	Oui
Robustesse	Très basse	Très haute	Moyenne
Sûreté	Basse	Haute	Moyenne
Modularité	Oui	Oui	Oui
Flexibilité	Non	Non	Oui
Extensibilité	Oui	Oui	Oui
Adaptabilité	Non	Non	Oui

Plusieurs architectures hybrides ont été proposées. Arkin [Arkin 87], avec l'architecture AuRA, et Payton [Payton 86], par exemple, proposent des modèles où le niveau inférieur est comportemental, mais l'addition d'autres niveaux hiérarchiques dote le système de la

capacité de planifier[2]. L'architecture AuRA (*Autonomous Robot Architecture*) incorpore en plus certaines inspirations de la neurophysiologie à propos du système endocrinien pour choisir le comportement dominant.

Selon notre point de vue, un élément centralisé de raisonnement global de haut niveau est nécessaire pour les robots avec le degré d'autonomie et de programmabilité qu'on souhaite traiter dans ce travail. Cet agent de prise de décisions stratégiques doit avoir accès à toute la connaissance de haut niveau disponible pour raisonner à propos de l'état global de l'environnement. En outre, l'unité de contrôle intelligent doit être indépendante de l'organisation et de la façon d'interagir des autres composants du système. Il doit agir plus comme un processeur intelligent d'informations symboliques que comme un générateur de commandes.

Cela préconise la génération des actions nécessaires de commande par les activités qui interagissent avec l'environnement elles-mêmes, découplant les actions d'assimilation et de raisonnement de haut niveau des réponses réactives de bas niveau. La partie réactive est plutôt guidée que contrôlée par la partie délibérative. En conséquence, les sous-systèmes réactifs, qui tournent continûment et de façon parallèle et indépendante du sous-système délibératif, peuvent répondre rapidement à des événements externes (comme des obstacles qui s'approchent).

Pour assurer la réactivité "intelligente" du système aux changements dans l'environnement, le trafic d'information entre les modules de perception et de raisonnement doit être maintenu au minimum afin d'éviter des retards dûs à la communication. Cela suggère que l'information à échanger doit être d'un haut niveau d'abstraction, c'est-à-dire, des données traitées par opposition à des données brutes. Pour cela, les modules de perception doivent traiter les données brutes et extraire les caractéristiques pertinentes de l'environnement. Ces "extracteurs de caractéristiques" doivent fournir l'information en temps réel pour garantir que les décisions de contrôle seront basées sur la connaissance la plus récente.

Un type d'architecture hybride qui peut satisfaire un bon nombre des exigences présentées, et qui est donc devenu très populaire [Hasemann 95], sont les architectures à trois niveaux. Ces systèmes emploient normalement trois niveaux d'abstraction: un niveau délibératif, un niveau de séquencement et un niveau réactif.

1. Le niveau délibératif utilise des représentations classiques de l'intelligence artificielle et techniques de raisonnement comme planification temporelle, ordonnancement (*scheduling*) et allocation de ressources. Les activités de ce niveau correspondent à la planification de stratégies de long terme et aussi à des éventuelles modifications du plan. Ce niveau dépend d'une connaissance très abstraite et de techniques de raisonnement très sophistiquées, et il est le domaine typique des planificateurs capables de faire usage d'une connaissance extensive du domaine d'application.
2. Le niveau de séquencement fait apparaître une sorte de planificateur réactif qui sélectionne une tactique appropriée selon des règles dépendantes du contexte. Une

2. Arkin et Payton sont normalement cités parmi les pionniers du paradigme comportemental. À notre avis, cependant, leurs propositions relaxent certaines des contraintes de base de ce paradigme, comme l'inexistence d'une représentation globale centralisée; ce qui nous fait préférer les classer comme des architectures hybrides.

tactique est un ensemble pré-écrit d'actions ordonnées, riche en structure (conjonctions et disjonctions d'actions, récursion, hiérarchie d'actions, etc.). Le niveau de séquencement choisit une tactique et l'exécute, ce qui mène à l'activation et à la terminaison d'activités du niveau réactif et implique le déclenchement de surveillances, la reconnaissance des conditions de terminaison et des erreurs d'exécution, etc.
3. Le niveau réactif assure la transition entre le raisonnement symbolique et la commande numérique non symbolique et combine les activités séparées.

1.5 Concepts généraux de l'architecture LAAS

L'architecture LAAS a été conçue pour permettre au robot d'accomplir ses tâches avec un comportement autonome et intelligent, qui combine la délibération et la réactivité [Chatila 95]. Pour obtenir aussi bien une capacité d'anticipation qu'un comportement en temps réel acceptable, le système a été décomposé en deux niveaux principaux:

Niveau fonctionnel – Il regroupe les capacités de perception et d'action du robot et ses réactions réflexes. Il inclut les fonctions de traitement des données et les boucles d'asservissement.

Niveau décisionnel – Ici sont regroupées les prises de décision et les capacités de planification et d'anticipation du système. Le niveau décisionnel peut être à son tour divisé en sous-niveaux hiérarchisés, où les sous-niveaux inférieurs manipulent des représentations des actions qui sont plus proches des conditions d'exécution.

Toutes les 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. Un *exécutif* centralise le flux de contrôle entre les modules et le niveau décisionnel et gère en temps réel l'exécution des fonctions.

La couche fonctionnelle, outre cet aspect architectural, doit également satisfaire aux contraintes inhérentes aux systèmes informatiques temps réel distribués. En particulier, elle doit offrir des mécanismes d'activation/désactivation, de synchronisation et de communication efficaces. Les modules et l'exécutif se chargent de fournir ces services.

Planifier est le mot clé pour le niveau décisionnel. Mais un robot autonome interagit en permanence avec l'environnement, et la planification doit être faite en parallèle avec les autres activités. Comme la planification requiert normalement une quantité de temps plus grande que la dynamique imposée par l'occurrence d'un événement, l'architecture LAAS préconise le contrôle du niveau fonctionnel par un système délibératif qui a un temps de réaction borné pour une première réponse après la réaction réflexe. Pour cela, le (ou les sous-niveaux du) niveau décisionnel doit (doivent) être conçu(s) selon ce qui a été appelé le *paradigme superviseur - planificateur* [Alami 93].

1.5.1 Le paradigme superviseur - planificateur

Chaque niveau décisionnel[3] doit être composé de deux entités indépendantes: un *planificateur*, qui produit la séquence d'actions nécessaires pour accomplir une tâche ou atteindre un but, et un *superviseur*, qui interagit avec le niveau inférieur pour contrôler l'exécution du plan et réagir à des événements qui se produisent (fig. 1.4).

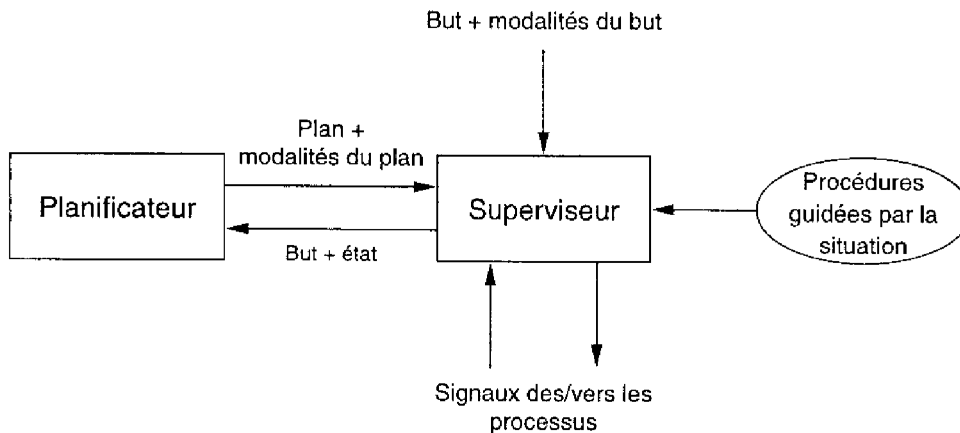


FIG. 1.4 – *Paradigme d'organisation du niveau décisionnel*

Le planificateur reçoit une description de l'état du monde et un but, à partir desquels il produit un plan. Pour que le système soit robuste, le plan, plus qu'une simple séquence d'actions, doit contenir un ensemble de modalités d'exécution. Ces modalités peuvent être:

- des contraintes ou des indications à être utilisées par un éventuel niveau de planification inférieur; ou
- une description des situations à surveiller, avec les réactions conséquentes à adopter:
  - des action réflexes immédiates;
  - des actions "locales" de correction (qui ne mettent pas en cause l'essence du plan principal); ou
  - une requête de replanification.

Le superviseur interagit avec les autres niveaux et avec le planificateur. Son activité consiste à surveiller l'exécution du plan à travers les signaux qui lui arrivent des autres niveaux. En cas de déclenchement d'une des surveillances qui ont été mises en place pour le contrôle d'exécution du plan, il doit prendre en temps réel les décisions qui s'imposent (selon les modalités permises par le plan).

Dans chaque niveau, le superviseur est le seul responsable de la communication avec les autres niveaux (et donc avec l'environnement). Pour que le système soir réactif, il ne doit utiliser que des algorithmes qui ont un temps de réponse borné et compatible avec

3. Pour être précis, on devrait parler de "chaque sous-niveau du niveau décisionnel", ce qu'on ne fera plus dorénavant par des raisons de concision.

la dynamique des processus qu'il contrôle (tout autre algorithme de délibération étant destiné au planificateur). Les actions qu'il accomplit sont soit prévues dans le plan, soit choisies, selon le contexte, dans un ensemble de procédures prédéfinies, indépendantes du plan.

1.5.2 Analyse comparative

Par rapport aux architectures hiérarchisées, l'architecture LAAS garde la structure d'assimilation et abstraction progressive des données et le raisonnement de haut niveau qui prend en compte toute l'information disponible. La faible réactivité, caractéristique des hiérarchies strictes, est évitée par la possibilité donnée à chaque niveau, s'il sait comment le faire, de "court-circuiter" les niveaux supérieurs pour agir directement sur le système (fig. 1.5).

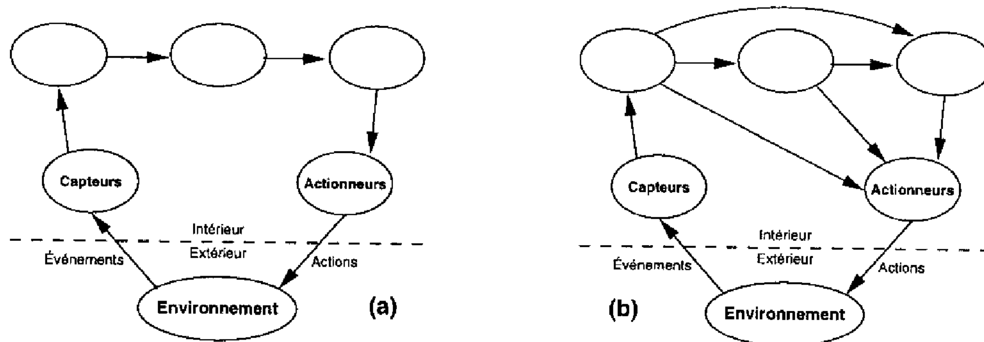


FIG. 1.5 – *Les architectures hiérarchisées (a) par rapport à l'architecture LAAS (b)*

Une bonne caractéristique des architectures comportementales est le fait de permettre à un agent local de prendre le contrôle du robot pendant une période de temps pour exécuter, de façon indépendante et avec des données locales, sa tâche spécifique. L'architecture LAAS retient d'une certaine façon ce concept, parce que quelques-uns de ses modules (mais pas tous) dotent le robot d'un comportement qui peut être exhibé de manière autonome[4]. La différence majeure réside dans le fait que la décision de prendre le contrôle du robot ne vient pas du module, mais d'un agent de raisonnement centralisé (en l'occurrence, le niveau décisionnel à travers l'exécutif) qui choisit le meilleur comportement pour le robot en fonction de sa tâche globale et donne au préalable au module approprié l'accès aux capteurs et actionneurs nécessaires (fig. 1.6 page suivante). Cette configuration permet d'avoir une excellente réactivité sans renoncer à la centralisation des prises de décision.

Finalement, le niveau fonctionnel de l'architecture LAAS adopte une organisation centralisée autour de l'exécutif: cela garantit une bonne évolutivité et un développement flexible pour cette partie du système, où se concentre une bonne partie de l'effort de mise

4. Par exemple, le module d'évitement local d'obstacles, présent en plusieurs mises en œuvres de l'architecture LAAS

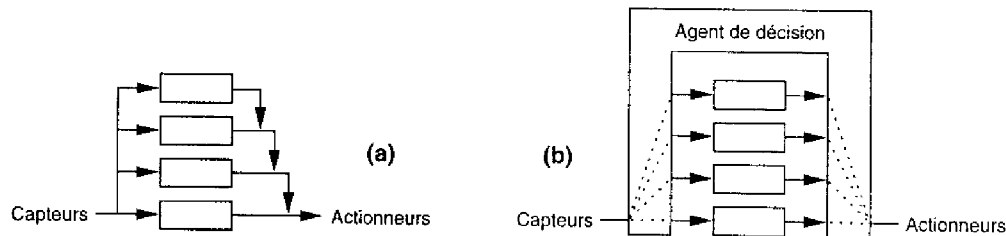


FIG. 1.6 – *Les architectures comportementales (a) par rapport à l'architecture LAAS (b)*

en œuvre du robot. Cette centralisation, cependant, n'impose pas de surcharge au système: il n'y a pas de duplication de données, qui sont stockées dans des zones de mémoire accessibles en lecture à tous les modules, ni des algorithmes redondants qui tournent en parallèle, vu que les décisions sont prises par planification et anticipation et non par choix de la réponse la plus adéquate. La probabilité que le module central devienne un goulet d'étranglement du système, même si elle doit toujours être prise en compte, a été réduite par trois mesures (fig. 1.7):

1. La grande majorité des données n'est pas centralisée.
2. Même le flux de contrôle n'est pas entièrement centralisé: les modules, sous certaines conditions, peuvent envoyer des requêtes directement à un autre module, sans passer par l'exécutif (il doit seulement en être informé).
3. On garantit pour l'exécutif un temps de réponse borné et assez petit par rapport à la dynamique du système.

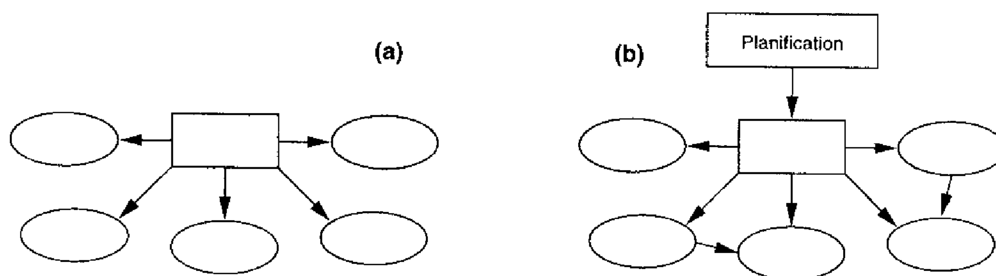


FIG. 1.7 – *Les architectures centralisées (a) par rapport à l'architecture LAAS (b)*

Il reste le fait que, étant donné la structuration du système en niveaux, il est nécessaire de définir préalablement une interface et un protocole de communication entre les composants avant de pouvoir les mettre en œuvre. Cette définition peut restreindre le développement ultérieur des modules si les stratégies adoptées ne sont pas assez flexibles, ce qui nous a conduit, dans ce travail, à approfondir l'étude du rôle et des mécanismes de communication entre les divers composants de l'architecture.

1.6 L'architecture LAAS à trois niveaux

Les concepts généraux de l'architecture LAAS peuvent conduire, au moment de la conception effective, à des mises en œuvres différentes selon le type de robot à être contrôlé. Pour les robots mobiles autonomes, il est normalement adopté une organisation en trois niveaux, représentée dans la fig. 1.8 et qui sert de base aux travaux de ce mémoire: deux niveaux décisionnels (niveau tâche et niveau mission) et un niveau fonctionnel, composé d'un exécutif et d'un ensemble de modules. Les deux niveaux supérieurs sont construits en respectant le paradigme superviseur - planificateur.

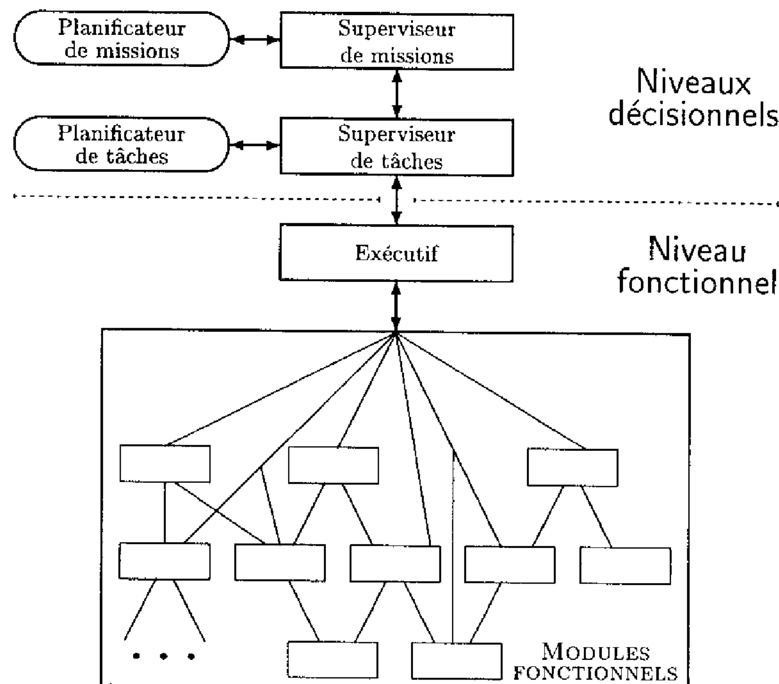


FIG. 1.8 – *L'architecture LAAS à trois niveaux*

1.6.1 Le niveau mission

Ce niveau est basé sur les techniques générales de planification d'actions. Le LAAS a développé le système IxTeT (*Indexed Time Table*) [Ghallab 89] de planification temporelle, qui peut raisonner sur des relations numériques et symboliques entre instants temporels: il produit des plans formés par un ensemble de tâches partiellement ordonnées et par des contraintes temporelles, comme les durées minimale et maximale ou des synchronisations avec des événements externes. Les tâches (*tasks*) sont le niveau minimum de description connu par IxTeT, et correspondent aux actions élémentaires fournies par le niveau tâche de l'architecture de contrôle.

Le superviseur de plans qui fait partie du système IxTeT correspond au superviseur du niveau mission de l'architecture de contrôle (fig. 1.8 page ci-contre). Il a la charge de surveiller l'exécution des tâches impliquées dans un plan, vérifier que les tâches produisent vraiment les effets attendus et réagir en cas de désaccord. Le contrôle d'exécution dans ce niveau sera globalement assuré par le superviseur de plans, qui organise les tâches et les envoie vers le niveau inférieur (éventuellement après une étape de raffinement). Dans ce mémoire on ne traite pas particulièrement les questions liées au niveau mission ou à ses relations avec le niveau tâche: une introduction à ces problèmes peut être trouvée dans Chatila *et al.* [Chatila 92].

1.6.2 Le niveau tâche

Le deuxième niveau décisionnel reçoit des tâches qu'il transforme en séquences d'actions, appelées procédures (*scripts*), composées par des actions élémentaires du robot. Il supervise l'exécution des procédures en étant réactif à des événements asynchrones et à des changements dans les conditions de l'environnement. La planification à ce niveau est plutôt un "raffinement" des tâches élémentaires du niveau mission, qui sont adaptées en tenant compte des modalités spécifiées dans le plan et du contexte d'exécution.

Au niveau tâche il n'y a pas de planificateur général: le raffinement est en fait une sélection (très dépendante du contexte) d'actions dans une structure précompilée, ce qui garantit une réponse en temps borné. Mais il ne faut pas voir le niveau tâche comme un simple interpréteur qui exécute un plan réactif (plan + modalités) produit par le niveau mission. En effet, le planificateur de tâches fait réellement des évaluations et prend des décisions sur la manière que les actions seront exécutées. En outre, le superviseur de tâches peut décider qu'une replanification est nécessaire, et dans ce sens il peut influencer le comportement du niveau mission.

Le niveau tâche de l'architecture de contrôle, dans les réalisations actuelles, est fortement lié à PRS (*Procedural Reasonning System*) [Georgeff 86, Ingrand 92], le système de raisonnement procédural utilisé pour sa mise en œuvre. PRS, qui sera mieux présenté dans le chapitre 4, est composé d'un ensemble d'outils et de méthodes pour représenter et exécuter des plans et des procédures. Ces procédures sont des séquences conditionnelles d'actions et/ou de buts qui peuvent être exécutées (les premières) ou postés (les derniers) pour accomplir des buts précédemment postés ou pour réagir à des situations particulières. Un but posté peut requérir l'exécution d'une nouvelle procédure (choisie selon le contexte) qui peut, à son tour, poster des nouveaux buts.

Le niveau mission demande une exécution au niveau tâche en postant le but correspondant. Les éventuelles modalités d'exécution de cette tâche seront communiquées soit comme des paramètres du but posté, soit en écrivant dans la base de données de PRS. On peut avoir plusieurs procédures qui essaient d'accomplir un même but ou qui réagissent à l'apparition du même fait dans la base de données: le système choisira celle (ou celles) dont les préconditions sont satisfaites au moment de l'exécution. S'il y a plusieurs procédures applicables, le choix sera fait soit par les règles intrinsèques de PRS soit par des procédures spécialisées définies par le concepteur, les meta-procédures. Un but ne sera considéré comme non atteignable que lorsque toutes les procédures applicables auront été

essayées et auront échoué.

Le système PRS fournit les deux composantes nécessaires au niveau tâche dans l'architecture de contrôle (fig. 1.8 page 20): la supervision et la planification.

- L'aspect planification réside dans le choix dépendant du contexte de la procédure à appliquer. La syntaxe et le mode de fonctionnement de PRS permettent à chaque procédure de ne représenter qu'un plan partiel: elle contient les tests qui conditionnent le postage de nouveaux sous-buts, mais laisse à l'interpréteur (et aux meta-procédures) le choix de la procédure la plus adéquate pour essayer de satisfaire chaque sous-but posté.
- L'aspect supervision est assuré par la réactivité intrinsèque du système: l'exécution d'une procédure est questionnée à chaque pas par rapport au contenu de la base de données. En outre, les procédures déclenchées par des faits permettent de prendre en compte de façon asynchrone et en parallèle avec d'autres activités l'occurrence de certains événements: sous certaines conditions pas très contraignantes, C-PRS peut garantir une limite supérieure pour son temps de réaction.

1.6.3 L'exécutif

L'exécutif, présenté en détails dans le chapitre 3, fonctionne comme une interface entre le niveau décisionnel et les modules. Il est purement réactif (il n'y a aucune planification) et contrôle certains aspects du fonctionnement des modules selon des règles prédéfinies. Ses fonctions principales sont:

- Arbitrer les conflits entre fonctions de modules différents.
- Déclencher et surveiller l'exécution des fonctions des modules.
- Accomplir des actions réflexes.
- Masquer plusieurs aspects du fonctionnement des modules.
- Décrire l'état d'utilisation des ressources non partageables du robot.

L'exécutif peut être représenté par un ensemble d'automates à états finis, et sa mise en œuvre doit par conséquent être faite avec des outils adaptés à la description de ce type de système. Les règles de transition entre états de l'automate sont fondamentalement basées sur la connaissance que le concepteur possède sur le mode de fonctionnement du robot: il parait donc approprié qu'un système de représentation de la connaissance basé sur des règles soit adopté pour le décrire.

Mais la position occupée par l'exécutif dans l'architecture de contrôle impose certaines contraintes temporelles, comme un temps d'exécution borné et réduit, qui ne sont normalement pas respectées par les systèmes de développement basés sur des règles. La solution que nous avons adopté consiste à utiliser Kheops [Ghallab 88] pour mettre en œuvre certaines parties de l'exécutif. Ce système, développé au LAAS, permet de transformer un ensemble de règles propositionnelles en un réseau décisionnel optimisé au comportement déterministe. Par compilation des règles, Kheops produit un arbre de décision équivalent à la base de connaissances et qui garantit l'obtention, en un temps borné et optimal, d'un et d'un seul résultat pour chaque entrée: sa profondeur caractérise formellement le temps

de réaction maximal. Dans l'exécutif, l'automate produit par Kheops est activé à chaque occurrence d'un événement pertinent par un système externe.

1.6.4 Les modules

Les modules, qui sont bien décrits par Fleury [Fleury 94], doivent réagir à des requêtes externes d'un client et, après traitement et/ou exécution, retourner un rapport vers ce client, utilisant un protocole de communication du type client/serveur, avec requêtes et répliques. Si nécessaire, pendant l'exécution d'une fonction, un module peut envoyer (sous certaines conditions) des requêtes aux autres modules. Une zone de mémoire partagée, appelée un *poster*, est utilisée pour exporter des données par un module de façon à ce qu'elles puissent être lues par les autres modules.

On appelle activité une fonction d'un module en exécution. Une fonction peut avoir, à un instant donné, plusieurs instances actives, correspondant à autant d'activités. Une activité se commence par une requête d'exécution envoyée par son activité mère ou par l'exécutif. Si on considère les interactions entre les activités à un instant, les fonctions en exécution et leur connections sont représentées par un arbre. 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'activités filles: ainsi, l'interruption d'une activité doit entraîner l'interruption de ses activités filles.

Certaines activités doivent être présentes en permanence dès l'initialisation du système. Ces activités permanentes, peu nombreuses, concernent pour l'essentiel les asservissements de base du robot. Ces activités ininterruptibles sont directement sous le contrôle de l'activité à la racine de l'arbre.

Les activités matérialisent les actions atomiques[5] que le robot peut accomplir. À la fin de leur exécution, les fonctions émettent des rapports logiques: un rapport d'exécution, qui concerne l'état final de l'exécution (`OK`, `INTERRUPTED`, `ERROR`, etc.) et qui va être utilisé principalement par l'exécutif, et une description symbolique du résultat de l'exécution (quand cela s'applique) qui sera transmise au niveau décisionnel. Le résultat "brut" (sous format numérique), quant à lui, sera stocké soit dans un poster, soit dans des variables au niveau de l'exécutif.

Un module contient normalement des fonctions qui manipulent les mêmes données ou les mêmes ressources, ce qui peut conduire à des situations de conflit. Dans le cas d'une nouvelle requête incompatible avec une autre qui s'exécute, le module interrompt l'activité en cours et commence la nouvelle. La requête interrompue va occasionner une réplique finale qui signale l'événement. On verra par la suite que cette politique de donner toujours la priorité à la requête la plus récente n'est pas une politique globale de l'architecture de contrôle, mais simplement une conséquence de la centralisation des décisions dans le niveau décisionnel, qui a besoin de ce comportement du niveau fonctionnel pour pouvoir appliquer sa stratégie globale de résolution de conflits.

L'organisation des modules n'est pas fixe. Leur interaction, gérée par l'exécutif, dépend de la tâche qui s'exécute et du contexte de l'environnement: on peut, par exemple, avoir un

5. Atomiques parce qu'elles sont indivisibles, et non parce qu'elles sont nécessairement simples, courtes ou élémentaires

module qui se charge de maintenir le robot sur une trajectoire donnée, mais le générateur de cette trajectoire peut changer selon la situation. Cette importante propriété permet d'obtenir un comportement flexible et non systématique du robot.

Dans la fig. 1.9 on présente un arbre d'activités qui illustre cet établissement dynamique de liens entre les activités: on souhaite exécuter une trajectoire sans heurter d'obstacle mais sans s'éloigner trop de la trajectoire de référence pendant leur évitement. Pour ce faire, trois activités[6] se dérouleront en parallèle: la génération de la trajectoire, l'évitement d'obstacles et la surveillance de la déviation. Ces activités se redécomposent elles-mêmes en sous-activités. Ainsi, l'évitement d'obstacles fera appel à deux activités filles: la lecture des ultrasons et la génération de la consigne pour l'asservissement. La figure indique aussi le flux des données dans cette situation, qui part des capteurs (odométrie et ultrasons) et du générateur de trajectoire (qui fournit la trajectoire souhaitée) vers les actionneurs (moteurs des roues).

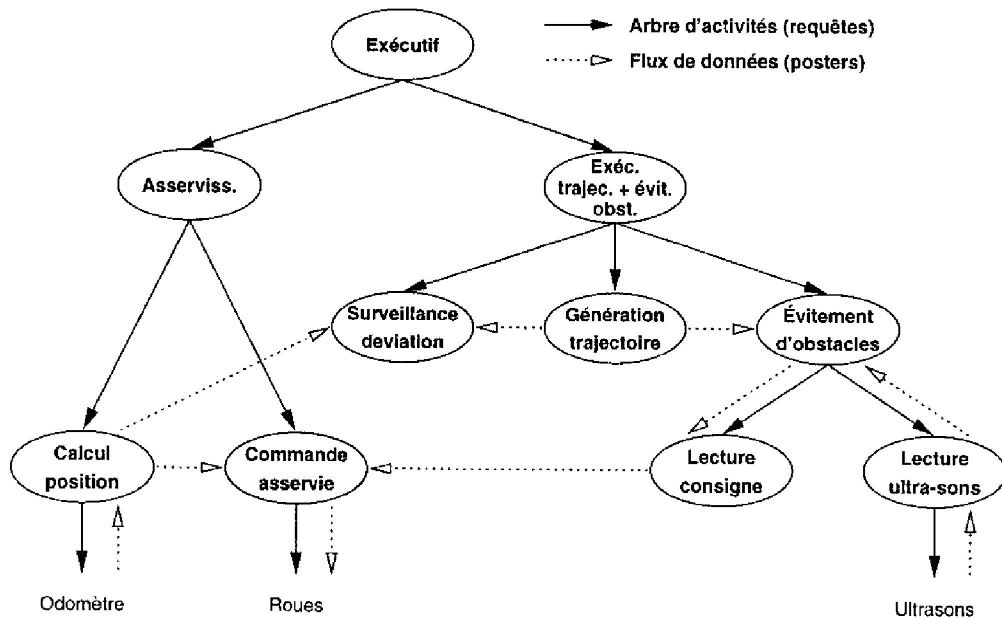


FIG. 1.9 – *Un exemple d'arbre d'activités à l'intérieur du niveau fonctionnel*

Le fait que les modules puissent envoyer des requêtes directement aux autres modules sans passer par l'exécutif ne pose pas de problème pour le maintient de l'information sur l'état des modules, si l'on respecte certaines règles présentées dans la section 2.6. L'exécutif, quand il déclenche une activité, prend déjà en compte les requêtes que cette activité va envoyer pour actualiser l'état des modules et résoudre les éventuels conflits.

Quand une activité n'est pas sous contrôle direct de l'exécutif, certaines des fonctions de ce composant de l'architecture devront nécessairement être remplies par le module

6. En plus des activités permanentes d'asservissement de la position du robot.

client. Entre autres, les tâches suivantes incombent à l'activité mère:

- Utiliser le protocole de communication client/serveur pour lancer l'activité fille et veiller à ce que toutes les répliques soient bien lues et interprétées. Cet accompagnement doit être fait sans utiliser des procédures qui bloquent (et rendent donc ininterruptible) l'activité mère.
- Interrompre toutes les activités filles si jamais elle-même est interrompue.
- Donner un sens aux informations qui lui sont envoyées par les activités filles pour envoyer les bilans appropriés à son propre client. Par exemple, si une activité fille s'interrompt à cause d'une erreur due au système, l'activité mère peut arrêter son exécution et signaler la même erreur ou, si cette possibilité avait déjà été prévue par le programmeur, poursuivre le traitement avec une autre méthode.

1.6.5 La distribution des données

Comme il est d'usage dans ce genre d'architecture, plus le niveau est élevé dans la hiérarchie, plus les données traitées sont abstraites. Les données numériques brutes résident dans le niveau fonctionnel, alors que le niveau décisionnel s'occupe principalement des données abstraites (des variables dénombrables, entières ou symboliques, pour la plupart évoluant dans des domaines finis).

A l'intérieur du niveau fonctionnel, les données à forte consommation de mémoire de stockage (images, trajectoires, représentations du terrain, etc.) résident soit dans la structure de données interne d'un module, s'ils n'ont qu'un intérêt local, soit dans un poster si d'autres modules doivent les consulter. Les données de taille plus réduite peuvent aussi être matérialisées comme des variables dans l'exécutif, principalement si le niveau décisionnel a besoin de les manipuler.

Deux préoccupations majeures dans une hiérarchie de données sont la duplication et la cohérence de l'information entre les divers niveaux. Avec cette séparation basée sur leur complexité, on évite la répétition des données, mais il reste à garantir leur cohérence: il faut être sûr que les données du niveau décisionnel soient effectivement une abstraction (après une éventuelle assimilation et fusion) des données du niveau fonctionnel et qu'il n'y a pas de conflits d'informations entre les deux niveaux.

Une stratégie qui a été adoptée pour garantir la cohérence consiste à rendre chaque fonction de module responsable de l'abstraction des données qu'elle génère. Chaque fonction qui crée des données, quand cela s'applique, doit fournir une représentation de cette information sous un format utilisable par le niveau décisionnel. Par exemple, une fonction qui cherche un objet dans une image doit retourner non seulement la position de l'objet mais aussi une description logique du résultat de la recherche (`TROUVÉ`, `PAS_TROUVÉ`, etc.) avec les informations dont le niveau décisionnel peut avoir besoin pour les prochaines décisions à prendre.

1.6.6 Un exemple d'exécution

Pour donner un aperçu de la démarche globale d'exécution, on présente le flux de contrôle et de données pour un exemple simplifié dans l'architecture LAAS. Le cas traité,

schématisé dans la figure 1.10, concerne un hypothétique robot d'exploration planétaire qui reçoit des missions de la base sur Terre pour les exécuter de façon autonome: on présentera un scénario nominal, sans erreur ni situation imprévue.

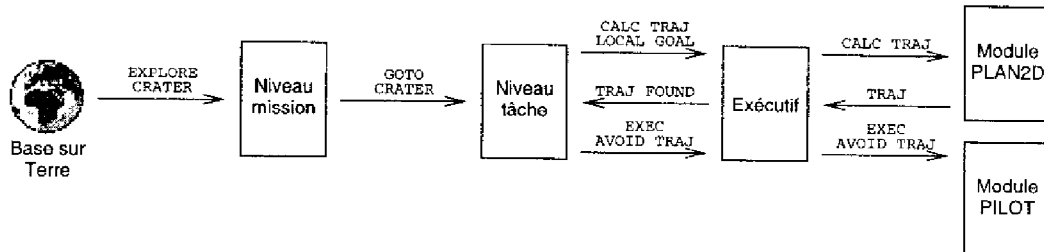


FIG. 1.10 – *Un exemple d'exécution de mission*

La mission reçue de la base pour une certaine journée consiste à explorer une région notable (disons une cratère) de la planète. Le niveau mission de l'architecture, à partir de cette information, établit un plan de mission pour la journée en tenant compte des contraintes spécifiques: les déplacements ne peuvent se réaliser que quand il y a de la lumière, les communications avec la Terre ne sont possibles que pendant certaines périodes de la journée, la récolte d'échantillons doit se faire quand la transmission des données est possible, etc. Une fois le plan (avec les modalités) établi, les actions correspondantes sont sollicitées au niveau tâche: pendant leur exécution, l'aspect supervision du niveau mission surveille si les bornes temporelles sont respectées et s'il n'y a pas d'erreur signalée.

À un moment donné, le robot devra se déplacer vers la position désirée: le niveau mission enverra le but, disons `(GOTO CRATER_21A)`, vers le niveau tâche. Il débute alors, dans le niveau tâche, le choix de la procédure la plus appropriée pour satisfaire au but désiré (`GOTO`), étant donné que chaque procédure a ses conditions d'applicabilité: par exemple, la procédure peut être différente selon le degré de fiabilité de l'information quant aux positions du robot et du but ou selon le type de terrain et la distance à parcourir. Supposons que la procédure adoptée exécute des boucles de l'algorithme suivant[7] jusqu'à arriver au but:

1. Une image vidéo est obtenue dans la direction du but: si la région a atteindre est reconnue dans l'image, sa position est réactualisée et le but final est déterminé; sinon, un but intermédiaire est choisi, selon des critères qui prennent en compte les possibilités de recalage et de visualisation de nouvelles régions des possibles sous-buts.
2. À partir d'une image laser 3D, le terrain est classifié et un mode de navigation est choisi. Le planificateur de trajectoire approprié est utilisé pour calculer une trajectoire jusqu'au sous-but (éventuellement le but final).
3. La trajectoire est exécutée avec un asservissement sur les positions calculées et une surveillance d'obstacles. À la fin, la position du robot est recalculée.

7. Cet exemple hypothétique s'inspire d'un algorithme qui a été effectivement mise en œuvre pendant l'expérience EDEN [Lacroix 94] faite avec le prototype de robot d'exploration spatiale *Adam*.

Pour ce qui concerne la communication entre le niveau tâche et l'exécutif et entre l'exécutif et les modules, on va se concentrer sur la période où le robot termine le calcul d'une trajectoire et commence à l'exécuter. La planification de trajectoire aura été lancée par une requête CALC_2DTRAJ_LOCAL_GOAL envoyée du niveau tâche vers l'exécutif. La procédure qui a émis cette requête reste bloquée[8] en attente du résultat de la planification de trajectoire. Quand le module responsable finit le calcul, l'exécutif envoie le résultat symbolique de l'exécution: 2DTRAJ_FOUND s'il n'y a pas eu d'erreur. Dans ce cas, la procédure du niveau tâche peut se poursuivre et une requête EXEC_AVOID_2DTRAJ est envoyée à l'exécutif.

Du point de vue de l'exécutif, quand la requête CALC_2DTRAJ_LOCAL_GOAL arrive, il fait une reconnaissance initiale pour déterminer quelle requête doit être envoyée, à quel module et avec quels paramètres. Dans ce cas, il s'agit d'une requête CALC_TRAJ, envoyée au module PLAN2D, et qui prend comme paramètres d'entrée les deux posters où sont stockés la position actuelle du robot et le sous-but à atteindre[9]. Après détection et résolution d'éventuels problèmes de conflits, l'exécutif envoie la requête et attend d'abord une réplique intermédiaire du module PLAN2D, qui signale qu'il l'a bien pris en compte, et ensuite sa réplique finale, avec l'état final et le résultat[10] de l'exécution. S'il n'y a pas eu d'erreur (l'état final d'exécution est OK), l'exécutif envoie la réplique finale de la requête CALC_2DTRAJ_LOCAL_GOAL vers le niveau tâche, qui contient le résultat fourni par la fonction CALC_TRAJ. Pour le traitement de la requête EXEC_AVOID_2DTRAJ, le déroulement est similaire: une requête EXEC_AVOID_TRAJ est envoyée au module PILOT, ayant comme paramètre d'entrée le poster où vient d'être stockée la trajectoire calculée par l'activité précédente.

Dans le module PILOT, la requête EXEC_AVOID_TRAJ va occasionner l'envoi d'autres requêtes à d'autres modules (comme AVOID, le module où se concentrent les fonctions d'évitement d'obstacles). Il sera donc établi un arbre d'activités similaire à celle de la figure 1.9 page 24, avec le flux de données correspondant:

1. l'activité de génération de trajectoire lit la trajectoire de référence calculée et enregistre périodiquement dans un poster des points de cette trajectoire, plus ou moins éloignés du point précédent selon la vitesse voulue;
2. l'évitement, avec la même périodicité, utilise l'information des capteurs pour vérifier qu'il n'y a pas d'obstacle sur le point en question: si oui, il enregistre dans son poster une position calculée par le module d'évitement pour une trajectoire tangentielle à l'obstacle; si non, la position du générateur de trajectoire est simplement copiée vers son poster;
3. la lecture de consigne désigne la position fournie par le module d'évitement d'obstacles comme nouvelle consigne pour l'asservissement en position du robot; et
4. la commande asservie compare la position actuelle avec la consigne et commande les roues pour mouvoir le robot si nécessaire.

8. Cela n'implique pas que tout le niveau tâche est bloqué: d'autres procédures ou même d'autres branches de cette procédure qui auront été lancées en parallèle continuent à s'exécuter normalement.

9. Ces deux valeurs doivent avoir été actualisées préalablement par des appels aux fonctions appropriées.

10. Il s'agit de la description abstraite du résultat de l'exécution: la structure de données qui contient la trajectoire calculée proprement dite sera stockée dans le poster correspondant

1.7 Conclusion

L'architecture LAAS a été définie au fil des ans, et sa configuration actuelle est le résultat de plusieurs travaux qui ont été réalisés au sein du groupe RIA. Notre contribution personnelle à cette œuvre collective se situe dans les aspects liés au contrôle d'exécution des activités, et se matérialise principalement autour de l'exécutif.

Ce composant de l'architecture avait déjà été prévu, au moins du point de vue conceptuel, dans quelques mises en œuvre de l'architecture, mais pas d'une façon systématique et avec une organisation standard, indépendante de la tâche. Son inclusion comme un élément individualisé a exigé la définition plus précise de certains aspects de l'architecture, qui sont présentés au long de ce mémoire:

- une meilleure définition du rôle et des fonctions des divers niveaux;
- un protocole de contrôle (activation, interruption et fin d'exécution d'activités) et d'échange de données avec les modules et le niveau décisionnel; et
- une distribution cohérente des données dans l'architecture.

Ces définitions ont permis de dégager les aspects du contrôle d'exécution qui sont indépendants de la tâche. Nous avons ensuite conçu une organisation interne de l'exécutif qui permet de mettre en place le contrôle d'exécution général, laissant au niveau décisionnel les aspects dépendants du contexte. Cette organisation, alliée à la répartition des données, a rendu envisageable une vérification plus formelle du niveau décisionnel.

Pour pouvoir présenter avec plus de détails l'exécutif, il est nécessaire d'aborder préalablement quelques concepts à propos des modules dans l'architecture LAAS. Dans le chapitre suivant on récapitule les principaux aspects liés à ce niveau de l'architecture et on discute des conséquences de notre stratégie de contrôle d'exécution sur les modules.

Chapitre 2

Les modules

Tout le contrôle d'exécution du robot se fonde, en dernière instance, sur le contrôle des activités des modules, qui sont les composants de base des fonctionnalités du système. Dans ce chapitre, nous résumons les travaux sur la conception et la réalisation des modules dans le cadre de l'architecture LAAS [Camargo 91, Fleury 94, Fleury 96] pour présenter en suite (sections 2.5 et 2.6) les conséquences de notre stratégie de contrôle d'exécution sur ce niveau de l'architecture.

2.1 Propriétés générales

On peut voir un module comme une collection de fonctions plus ou moins "intelligentes", avec des capacités de fonctionnement en temps réel, et comme un serveur qui, sur demande des clients, fournit certains services. Ils sont des entités capables d'exécuter des fonctions spécifiques en agissant de trois manières (qui peuvent se combiner) sur le robot et/ou sur l'environnement qui l'entoure:

- par production de données;
- en agissant sur des dispositifs physiques du robot (capteurs et actionneurs); et
- par activation d'autres modules.

Les services offerts par les modules sont matérialisés par les activités qui s'exécutent en son sein. Ces activités peuvent être classées en quatre catégories:

Les asservissements mettent en œuvre une boucle fermée de perception et d'action. Ce sont des activités périodiques dont les algorithmes déroulent toujours les mêmes séquences d'instructions et qui en mode nominal ne se terminent pas d'elles-mêmes.

Les filtres se distinguent des asservissement parce que, à la place d'agir sur l'environnement, ils produisent des données à partir des données d'entrée. Un filtre peut s'auto-terminer en cas de consommation de toutes les données d'entrée.

Les surveillances detectent (et se terminent à) l'occurrence d'un événement.

Les serveurs sont des activités asynchrones qui se terminent à l'issue du traitement.

Comme exemples de ces quatre catégories, on peut citer l'asservissement en position [1], la génération de points de consigne sur une trajectoire, la détection d'obstacles et le calcul d'un chemin entre deux points.

2.1.1 Le flux de contrôle

On accède aux différents services non permanents des modules au moyen de requêtes. Les requêtes, auxquelles on peut associer des paramètres d'exécution, permettent de démarrer, d'interrompre ou d'altérer les traitements. Elles peuvent être émises par un autre module ou par l'exécutif. La réception de la requête par le module établit une relation de type client/serveur (fig. 2.1), où le serveur n'a aucune connaissance *a priori* de l'identité de son client.

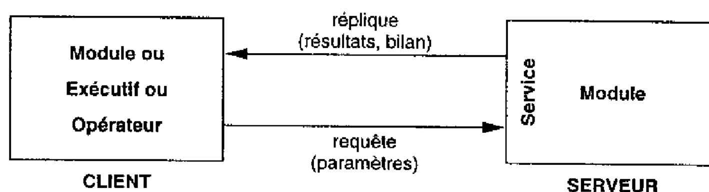


FIG. 2.1 – *Une relation client/serveur*

Un service se termine par l'émission en retour, du serveur vers le client, d'une réplique finale, qui indique le type de terminaison de l'exécution (normale ou pas) et clôt la relation client/serveur. Chaque réplique finale est aussi porteuse d'un bilan, qui qualifie la façon dont le traitement s'est déroulé, et peut retourner des résultats d'exécution. En mode nominal, le bilan d'exécution peut fournir une abstraction sur le résultat du traitement. Dans le cas contraire, il permet de caractériser le dysfonctionnement.

Les requêtes peuvent être de deux types: d'exécution et de contrôle. Les requêtes de contrôle ne démarrent pas d'activité: elles ont pour fonction d'accéder à des données internes du module, de modifier des paramètres d'exécution ou d'interrompre une activité. Le service associé est considéré de durée nulle et la réplique finale est par conséquent retournée immédiatement au client. Dans le cas d'une requête d'exécution, il y a le 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 toujours immédiat (par exemple, s'il y a des conflits avec d'autres activités qu'il faut interrompre), une réplique intermédiaire est émise vers le client après prise en compte d'une requête d'exécution. Contrairement aux répliques finales, les répliques intermédiaires ne retournent ni de bilan ni de résultat d'exécution.

Le niveau fonctionnel garantit que toute requête émise est réceptionnée et traitée par son destinataire au plus tôt et dans l'ordre d'arrivée. Au niveau des modules, il n'y a 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 l'exécutif, mais ne doit pas être vu comme une politique globale de résolution de conflits de toute l'architecture de contrôle.

1. L'asservissement en position est aussi un exemple d'activité permanente ininterruptible.

Dans le cadre d'une application intégrée, les conflits doivent être résolus par le niveau décisionnel, seul responsable des prises de décision du système et par conséquent de la définition d'une politique globale de priorités. Supposons, à titre d'exemple, qu'il soit nécessaire de décider de lancer ou non l'activité `A` alors que l'activité `B`, incompatible avec `A`, s'exécute. Si le niveau décisionnel, compte tenu de la tâche du robot, décide de ne pas lancer l'activité `A`, il suffit de ne pas envoyer la requête correspondante au niveau fonctionnel. Si, par contre, il décide de l'envoyer, cela signifie que l'activité a été considérée prioritaire et qu'elle doit être exécutée, avec l'interruption de `B`. La priorité accordée aux requêtes plus récentes au niveau des modules ne pose donc aucun problème (et est même nécessaire) à l'adoption d'une autre politique globale de priorités (qui peut et doit prendre en compte la mission générale du robot) dans l'architecture.

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. Comme ce contrôle d'interruption est local à chaque module, il permet d'interrompre directement une activité sans transiter par ses activités mères, qui en seront cependant informées par la réception des répliques finales. Une interruption n'est pas, en général, instantanée: l'activité va 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, libérer les ressources qui lui ont été allouées et réinitialiser des données en vue d'une prochaine activation.

2.1.2 Le flux de données

Les modules peuvent rendre visibles leurs données soit en répondant à une requête de contrôle que les sollicite, soit en les exportant dans un poster. Les posters sont des données structurées qui sont accessibles en lecture à tout élément de l'architecture de contrôle du robot mais en écriture seulement à son module propriétaire (fig. 2.2). Chaque module possède un poster de contrôle et un nombre variable, éventuellement nul, de posters d'exécution. Les posters de contrôle ont une structuration standard et contiennent des informations sur l'état courant du module (liste des activités, leur stade d'exécution, etc.). Les posters d'exécution sont spécifiques aux traitements mis en œuvre et sont mis à jour par les activités des modules.

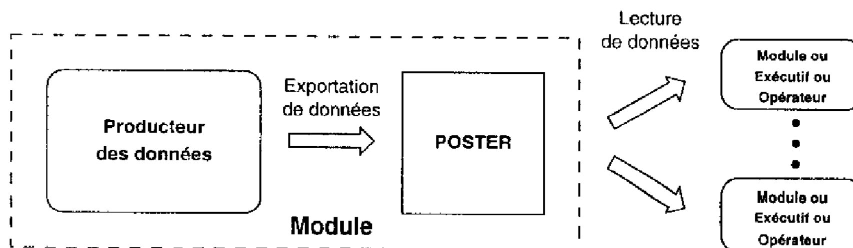


FIG. 2.2 – *Le flux de données: les posters*

On accède directement aux données du poster, au rythme souhaité et indépendamment du module propriétaire, ce qui les rend particulièrement intéressants pour:

- rendre publiques des données statiques volumineuses;
- disposer de données continuellement actualisés;
- transférer des données entre des activités avec contraintes temporelles différentes; et
- procéder à des transferts synchrones entre activités périodiques.

Pour ce qui concerne les informations locales du module, elles se localisent dans des bases des donnés appelées Structures de Données Internes (SDIs), accessibles en régime d'exclusion mutuelle. On distingue les données relatives au contrôle des activités et à l'état du module, qui composent la SDI de contrôle (SDI/c), des données relatives aux divers traitements, qui font partie de la SDI fonctionnelle (SDI/f). 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.

2.2 Architecture d'un module

Dans un module, on distingue le niveau de contrôle, qui gère les activités en fonction de requêtes, et le niveau d'exécution, où s'exécutent les activités. Les activités sont supervisées par le contrôleur de module. Ce contrôleur a, vis-à-vis de ces activités, quatre fonctions principales:

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

Les différentes étapes dans la vie d'une activité non permanente peuvent être représentées par un graphe d'état: le graphe de contrôle d'une activité (fig. 2.3 page suivante). Les cinq nœuds du graphe correspondent à des états de l'activité, qui sont généralement de durée non nulle, et les neuf arcs correspondent à des transitions considérées instantanées[2], imposées par le contrôleur (marquées ▭/-) ou signifiées par l'activité elle-même (marquées -/▭). 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 incompatibles antérieures sont interrompues.
- 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 l'amène dans l'état `EXEC`.

2. C'est à dire, avec un temps d'exécution compatible avec la constante temporelle de l'activité.

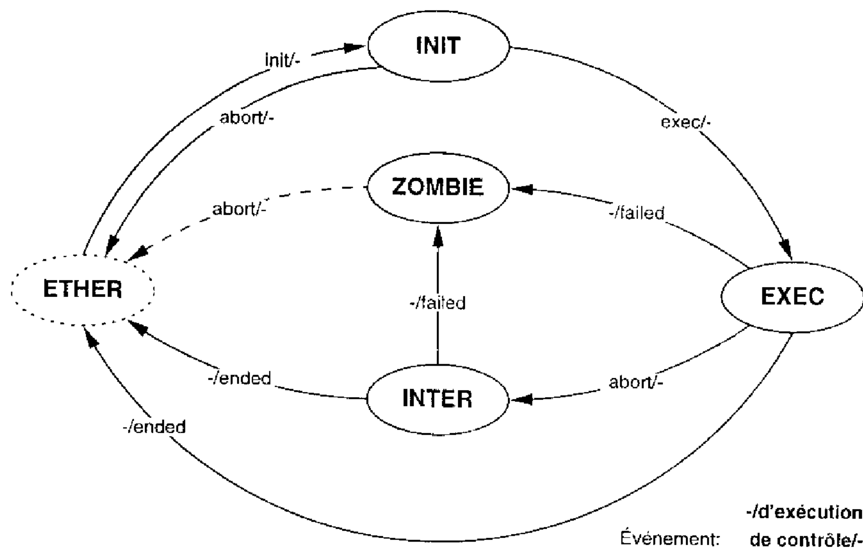


FIG. 2.3 – *Graphe de contrôle d'une activité*

- 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.

De façon générale, l'exécution d'une activité (correspondant à l'état EXEC dans le graphe de contrôle de l'activité) se décompose en différentes phases qui font chacune référence à une fonction. Ces fonctions sont des portions de code ininterruptibles, qui reçoivent le nom de codel (code élémentaire). La fonction exécutée durant l'état INTER, qui est lui-même ininterruptible, est également un codel. On peut voir l'état EXEC comme un macro-état qui encapsule des sous-états (fig. 2.4 page suivante). Ainsi une activité consiste en une succession de traitements prédéfinis, mais dont le séquencement est déterminé dynamiquement selon l'information retournée à l'issue de la phase précédente du traitement: l'activité transite dans un état X suite à un événement **-/x** spécifié à la fin de l'état précédent.

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'une même fonction (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.

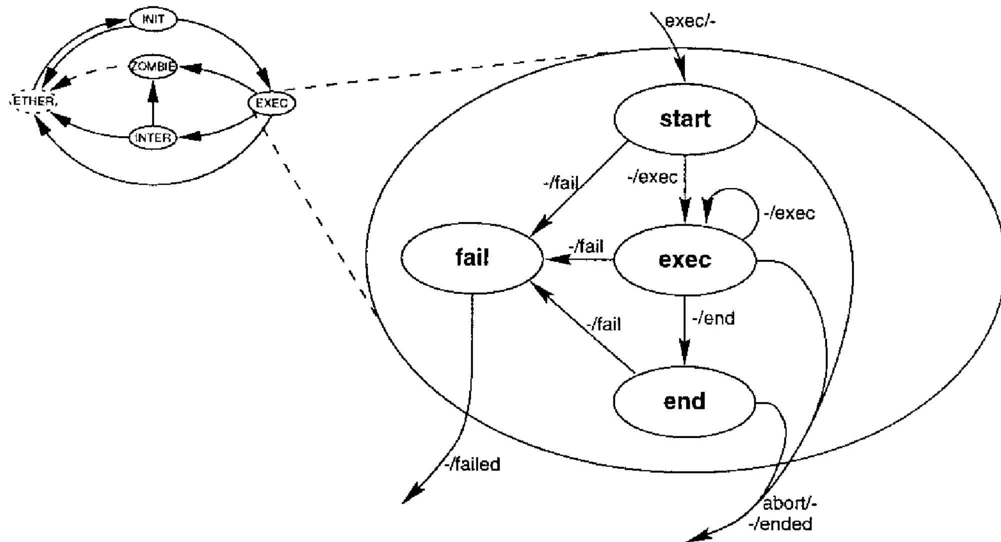


FIG. 2.4 – *Une instance du graphe de contrôle étendu d'une activité*

Si l'on vérifie les évolutions possibles du cycle de vie d'une activité non permanente, on constate qu'elle peut se terminer de quatre façons différentes:

1. la fin normale d'activités auto-terminantes;
2. la terminaison suite à un échec prévisible (insuffisance algorithmique, etc.);
3. l'interruption (par requête spécifique ou activation d'une activité incompatible); et
4. la terminaison suite à une défaillance extrinsèque de l'activité (comme une erreur due au système) qui l'a empêchée d'aller jusqu'au bout de ses possibilités algorithmiques.

Les deux premiers cas sont similaires du point de vue du module, étant donné que l'activité passe de son propre gré de l'état EXEC à l'état ETHER. Entre les deux situations, il n'y a que le bilan d'exécution, fourni par l'activité, qui change. Quant à la réplique finale, elle indique une terminaison régulière.

La terminaison à la suite d'une interruption n'a pas non plus des conséquences majeures au niveau du module, à part le fait qu'elle est signalée par une réplique finale spécifique et que le bilan d'exécution est généré par le contrôleur du module lui-même et non par l'activité. Du point de vue de l'activité, la spécificité de la situation réside dans le passage par l'état intermédiaire INTER, où sont effectués les traitements appropriés à la situation.

La terminaison suite à une défaillance extrinsèque (erreur du système d'exploitation, défaillance d'un capteur, etc.) est signalée par des répliques finales spécifiques, avec des

bilans d'exécution qui caractérisent plus précisément l'origine du problème. Les erreurs graves peuvent avoir des conséquences au niveau du module si jamais l'activité en question se met dans un état ZOMBIE, ce qui exigera une requête spécifique pour débloquer le module. Ce comportement a été adopté pour imposer au niveau décisionnel une prise en compte explicite des situations exceptionnelles.

2.3 Un exemple de réseau de modules

Le robot expérimental *Hilare* 2 (fig. 2.5) est équipé avec 2 roues motrices, 2 roues odométriques, une platine qui supporte deux caméras couleur et un télémètre laser 3D, un gyroscope directionnel pour la mesure du cap et une ceinture de 32 télémètres à ultrasons. La figure 2.6 page suivante présente l'organisation actuelle de son niveau fonctionnel, avec un exemple des relations client/serveur qui peuvent s'établir entre les modules.

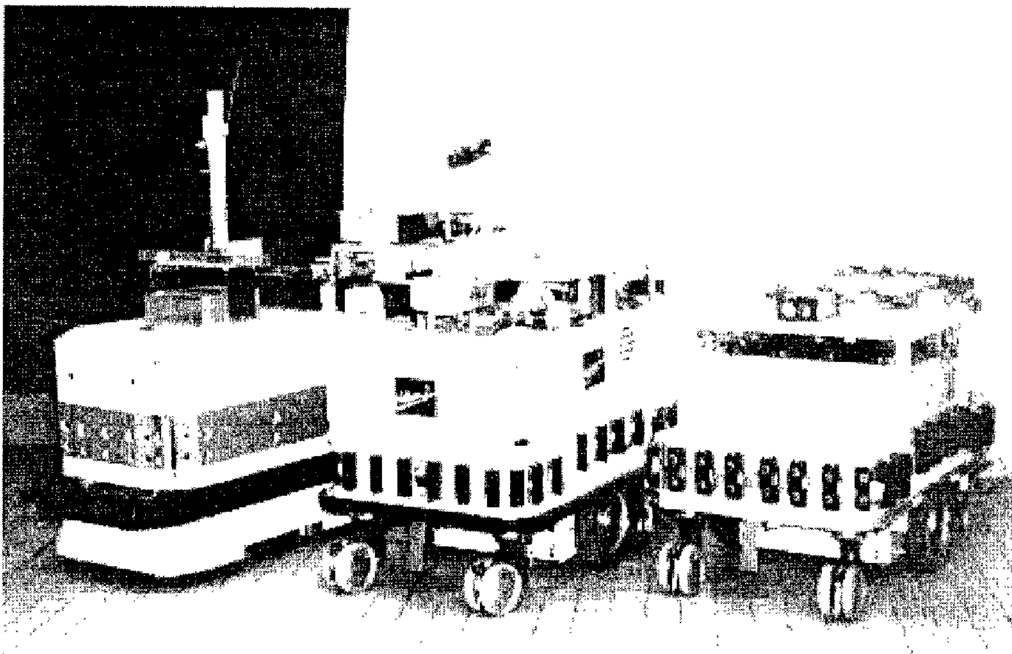


FIG. 2.5 – *La famille de robots Hilare, avec Hilare 2 au centre*

Le module US configure et active les télémètres à ultrasons, exporte des données de proximité et installe les surveillances d'obstacles; LOCO commande les roues motrices avec un asservissement en position, utilise le gyroscope et l'odomètre pour exporter la position calculée du robot et installe les surveillances qui concernent la position; PLATFORM contrôle l'orientation de la platine; VIDEO contrôle les caméras et l'acquisition des images; et TELE3D est utilisé pour acquérir des images 2D ou 3D avec le télémètre laser.

Au dessus de ces modules, on a les suivants: PILOT, qui génère des trajectoires dynamiques; AVOID, qui filtre les points de consigne pour éviter des obstacles à partir des

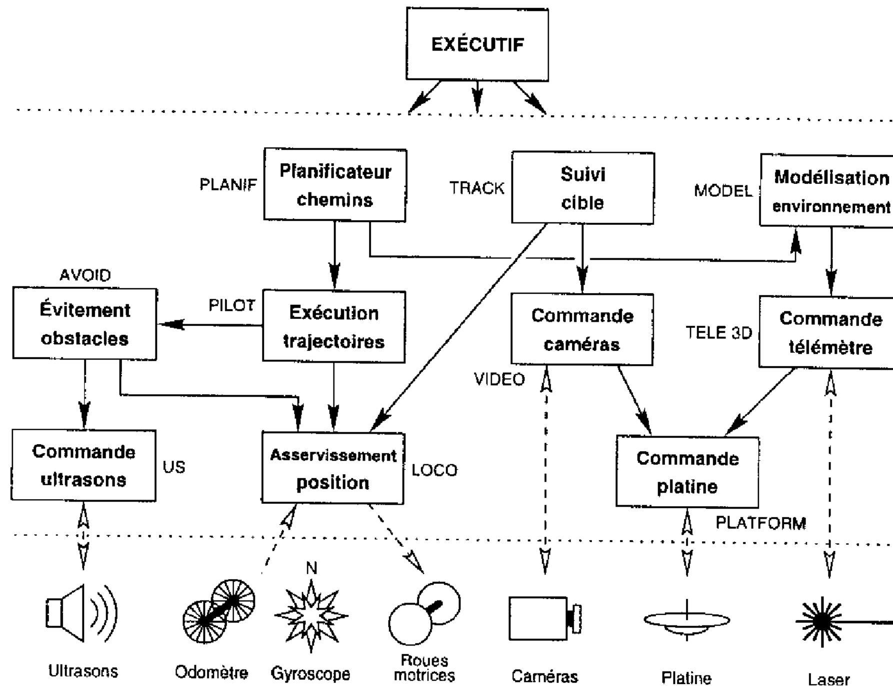


FIG. 2.6 – *Niveau fonctionnel du robot Hilare 2*

données de proximité; PLANIF, qui calcule des chemins sans collisions dans des environnements encombrés; TRACK, qui suit un objet à partir d'images vidéo; et MODEL, qui construit un modèle incrémental de l'environnement basé sur des segments.

2.4 Description formelle et génération automatique

Les modules ont une structure, un fonctionnement et des interactions bien définis dont la standardisation a permis d'élaborer un langage de spécification déclaratif permettant de les décrire de façon formelle. La génération automatique découle naturellement de cette description formelle. A cette fin, il a été développé un générateur de modules nommé G^{en}oM (*Generator of Modules*) [Fleury 96].

Le générateur se compose de deux éléments: le canevas d'un module générique et un analyseur syntaxique qui "remplit" le canevas à partir de la description formelle. Le canevas est un ensemble de programmes C, avec des mots-clé qui peuvent être substitués soit par une simple variable soit par le code d'une fonction. Il revient au concepteur du module de fournir sa description formelle et les codels pour les divers traitements (fig. 2.7 page suivante).

La description formelle permet de spécifier entièrement le 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

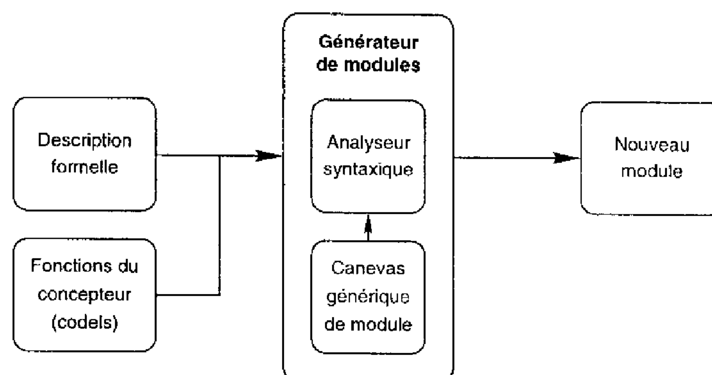


FIG. 2.7 – *La structure du générateur de modules*

bilans, etc. Elle emploie une syntaxe proche du langage C et comporte cinq parties:

1. La déclaration globale du module
2. La déclaration de la SDI fonctionnelle
3. La déclaration des requêtes
4. La déclaration des tâches d'exécution (et des activités permanentes associées)
5. La déclaration des posters d'exécution

2.4.1 Un exemple de description formelle

À titre d'exemple, on donne en suite un aperçu de la description formelle du module LOCO du robot *Hilare* 2:

Déclaration globale du module

La déclaration globale permet d'attribuer au module un nom (qui sera préfixé aux noms de tous ses composants) et un numéro d'identification et d'indiquer le type de sa SDI/f.

```
module loco{
  number:              700;
  internaldata:        LOCO_STR;
};
```

Déclaration de la SDI fonctionnelle

La déclaration des types des données contient toutes les déclarations en langage C pour les diverses structures utilisées dans le module de locomotion: représentations de la position courante, vitesse et accélération avec les matrices d'erreur associées et les paramètres utilisés pour décrire la géométrie du robot et l'algorithme d'asservissement.

Déclaration des requêtes

La requête locoGoTo est un exemple d'une requête de contrôle: elle écrit simplement une nouvelle valeur dans la SDI/f du module, ce qui a pour effet d'asservir le véhicule sur une nouvelle consigne et donc de le faire bouger.

```
request GoTo{
  type:              control;
  input:             ref::refStr;
  c_control_func:    controlRef;
  fail_msg:          BAD_REF_DATA;
};
```

La consigne est validée par la fonction controlRef (un codel), fournie par le concepteur. L'unique erreur possible est BAD_REF_DATA, au cas où la validation échoue.

La requête d'exécution pour le suivi de consigne (locoTrack) reçoit en argument (input) le nom du poster où doivent être lues les consignes. La phase d'initialisation d'exécution **start** vérifie l'existence du poster et la consigne est alors récupérée périodiquement (phase **exec**). Il ne peut y avoir qu'une instance de cette activité (incompatible_with): une seconde requête l'interromprait et poursuivrait le suivi de consigne sur un nouveau poster.

```
request Track{
  type:              exec;
  activity:          filter;
  input:             prName::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:          MANUAL_MODE, SHOCK,
         INVALID_REFERENCE, LOW_BATERY,
         POSTER_NOT_FOUND, EMERGENCY_STOP;
  incompatible_with: Track;
};
```

locoTrack 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 différentes possibilités de défaillance sont exprimées dans le champ fail_msg. La requête s'exécute dans le contexte de la tâche d'exécution PumpTask.

Déclaration des tâches d'exécution

La tâche d'exécution locoPumpTask se chargera d'une seule activité, le suivi de consigne. Elle a une période de 5 tics, ce qui dans la mise en œuvre actuelle équivaut à 25 ms, et une priorité relative 50.

```
exec_task PumpTask{
  period:            5;
  priority:          50;
  stack_size:        2000;
};
```

La tâche locoCmdTask a la priorité la plus élevée du système et ne s'occupe que des activités permanentes (calcul de la position puis asservissement). À chaque cycle, elle s'exécute avec un délai de 2 tics par rapport à locoPumpTask, ce qui permet au robot de s'asservir sur une consigne fraîchement actualisée.

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

Les activités permanentes du module ont la même période et s'exécutent séquentiellement: elles peuvent donc être associées dans la même tâche d'exécution (locoCmdTask) et intégrées dans le même codel (odoAndAsserv). Le codel initOdoAndAsser, invoqué au démarrage de la tâche, initialise la SDI/f.

Déclaration des posters d'exécution

La position du robot, son incertitude probabiliste et la consigne courante sont exportées dans le poster locoRobot. La mise à jour du poster se fait automatiquement à la fréquence de l'activité permanente d'asservissement odoAndAsserv.

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

2.5 Le protocole de contrôle

L'établissement et le suivi de la relation client/serveur sont basés sur des fonctions qui permettent au client d'envoyer la requête et de recevoir la(les) réplique(s) du module serveur. Si la requête désirée s'appelle *func*, du module *mod*, les deux fonctions de communication sont:

*modFunc*RqstSend
*modFunc*ReplyRcv

*modFunc*RqstSend prend en entrée les paramètres de la connexion (fondamentalement, les temps d'attente maximaux autorisés pour la réception des répliques intermédiaire et finale) et les éventuels paramètres d'entrée de la requête. Elle retourne par valeur le résultat de l'établissement de la connexion (OK ou ERROR) et en paramètre l'identificateur de la requête. *modFunc*ReplyRcv, avec l'identificateur comme paramètre d'entrée, retourne par valeur l'état du serveur (on l'appellera status) et en paramètre le bilan de l'activité (report) et les éventuels paramètres de sortie de la requête. Deux fonctions spécifiques permettent d'interrompre une activité du module: *mod*AbortRqstSend (dont un des paramètres d'entrée est l'identificateur de l'activité à interrompre) et *mod*AbortReplyRcv.

Jusqu'à présent, la bibliothèque client/serveur ne fixait pas précisément les messages à être échangés entre les partenaires, laissant à la charge du programmeur la définition des bilans qu'une fonction peut retourner après son exécution. Si cette démarche permet une grande souplesse en termes du type d'information échangé, elle rend plus difficile la tâche d'un élément de contrôle global, tel que l'exécutif, qui doit alors connaître chacun des dictionnaires spécifiques des activités qu'il gère.

L'autre extrême (la définition préalable des messages possibles de retour des fonctions) n'est pas envisageable: plusieurs fonctions doivent retourner des messages spécifiques, correspondant à des abstractions du résultat de leur exécution, qui ne peuvent être spécifiées que par leur concepteur. On a donc proposé une solution intermédiaire, basée sur une classification des terminaisons possibles pour une activité et sur un meilleur partage des informations entre status et report.

Un premier point concerne les situations de fonctionnement non nominales. On peut classifier les erreurs d'exécution d'une activité en deux catégories:

Les erreurs intrinsèques – plutôt que des erreurs, ce sont des situations d'échec où l'activité est allée jusqu'au bout de ses possibilités, mais l'algorithme mis en place

dans la fonction ne peut pas arriver à une réponse à cause d'une limitation intrinsèque de la méthode choisie et non d'un problème externe. Des exemples typiques sont les insuffisances algorithmiques, comme un terrain trop complexe pour un planificateur de chemins, ou les impossibilités réelles de satisfaire à une requête (par exemple, localiser un objet dans une image où il n'y apparaît pas).

Les erreurs extrinsèques – dans ce cas, la fonction a échoué suite à des problèmes qui lui sont extérieurs: mauvais paramètres d'entrée fournis par le client, erreurs liées au système d'exploitation (impossibilité de satisfaire à une demande d'allocation de mémoire ou de création d'un sémaphore, par exemple), au protocole d'établissement de relations client/serveur entre fonctions ou à l'accès aux posters, temps maximal d'exécution écoulé, etc.

L'information à transmettre au client de l'activité en cas d'erreur intrinsèque (comme en cas d'exécution nominale) est évidemment dépendante de l'activité. Pour les erreurs extrinsèques, et dans certaines autres situations, il serait cependant intéressant de fournir une information standardisée, qui permettrait d'utiliser des stratégies de reprise ou de terminaison contrôlée uniformes.

On peut donc définir un dictionnaire des types de terminaison envisageables pour une activité, à retourner (`status`) à la fin de l'exécution. Selon la valeur de `status`, le contenu du bilan (`report`) doit être interprété différemment. On présente dans le tableau 2.1 une des syntaxes possibles pour la communications entre clients et serveurs, qui a été utilisée dans ce travail[3].

TAB. 2.1 – *Dictionnaire des messages échangées entre clients et serveurs*

`status`	**Signification**	
	`status`	`report`
CS_ERROR	Erreur pendant la transmission de la requête ou des répliques	-
WAITING	En attente de réplique	Quelle réplique: la finale ou l'intermédiaire?
END	Fin nominale	Défini par le concepteur.
TIMEOUT	Temps maximal d'attente écoulé	Quel temps d'attente dépassé: le final ou l'intermédiaire?
INTERRUPTED	Activité interrompue	Cause de l'interruption: ABORT ou le nom de la requête conflictuelle.
FINAL_ERROR	Erreur extrinsèque	Code de l'erreur: erreur système, non activation d'activité fille, etc.
SERVER_ERROR	Erreur signalée par activité fille	Permet de récupérer l'erreur originale (enchaînement possible).
ZOMBIE	Activité en état ZOMBIE	Identificateur que permet de récupérer l'erreur originale.

3. Ce protocole n'a pas encore été incorporé à la génération automatique des modules, de sorte que, au niveau de l'exécutif, on simule partiellement son existence par conversion des informations qui sont effectivement envoyées par les modules.

2.6 Le contrôle d'exécution et les modules

Dans le contexte d'une application intégrée, la stratégie de contrôle d'exécution adoptée pour l'architecture globale impose certaines contraintes aux fonctions des modules. Elles concernent principalement l'utilisation des ressources non partageables du robot et l'accès direct (sans passer par l'exécutif) à des services d'autres modules. L'expression "utiliser une ressource" est employée dans le sens le plus large possible: toute activité qui peut créer un conflit avec une autre activité (même une autre instance de la même fonction en exécution) utilise une ressource non partageable du robot: il peut s'agir d'une ressource physique, comme une caméra, logique, comme un poster, ou même virtuelle, dans les cas où on n'explicite pas la raison exacte de l'incompatibilité entre les deux activités.

Dans notre stratégie, les conflits entre les fonctions sont analysés et résolus quand une requête arrive: le contrôleur du module (seulement pour les conflits locaux au module) et l'exécutif (pour tous les conflits), connaissant les ressources dont la nouvelle activité aura besoin, vérifient si les activités en cours les utilisent déjà. Comme l'allocation des ressources à une activité est faite *a priori* et reste valable pendant toute son exécution, les fonctions doivent respecter certaines règles pour que la gestion des conflits soit correcte et pour optimiser l'utilisation des ressources du robot. Il s'agit fondamentalement de bien choisir le niveau de granularité des fonctions, pour que les ressources non partageables du robot soient utilisées de façon persistante et prévisible.

Prévisibilité - Si une fonction utilise une ressource non partageable du robot, elle doit l'utiliser systématiquement (à chaque fois qu'elle est activée). Si jamais un traitement ne respecte pas cette règle, on peut le diviser en plusieurs fonctions différentes (fig. 2.8 page suivante), laissant au niveau décisionnel le choix d'appeler une d'entre elles. Ce choix, en plus d'augmenter l'efficacité du contrôle d'exécution, améliore la cohérence globale de l'architecture.

Persistance - Les fonctions doivent utiliser les ressources non partageables dont elles ont besoin pendant l'intégralité (ou au moins une bonne partie) de leur exécution: les traitements qui en principe n'obéissent pas à cette règle peuvent être subdivisés en plusieurs fonctions (fig. 2.9 page suivante) que le niveau décisionnel se chargera d'enchaîner.

Ces principes favorisent un partage optimisé des ressources du robot entre les activités, mais il peut s'avérer que dans certaines situations il ne soit pas faisable ou désirable de subdiviser suffisamment un traitement pour les respecter intégralement[4]. Dans ces cas, pour que la gestion des conflits soit correcte, la totalité des ressources dont la fonction peut avoir besoin devra lui être allouée pendant toute son exécution. Comme conséquence, l'activité correspondante aura une probabilité plus grande d'être interrompue, une fois que le principe de réactivité du niveau fonctionnel donnera la priorité à n'importe quelle autre nouvelle requête qui aura besoin d'une de ces ressources.

---

4. Par exemple, pour ne pas trop charger la communication entre les niveaux fonctionnel et décisionnel par l'envoi d'une série importante de requêtes.

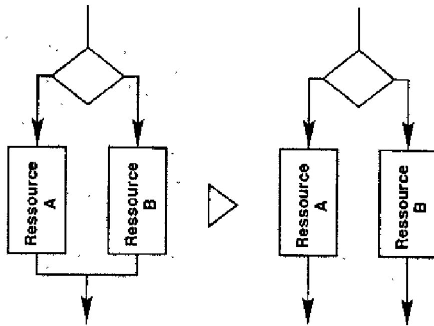


FIG. 2.8 – *Prévisibilité des fonctions*

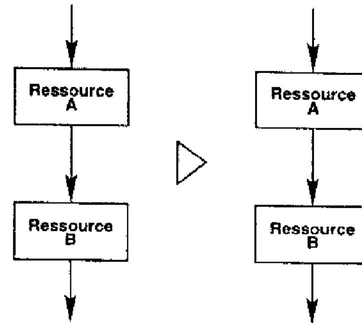


FIG. 2.9 – *Persistance des fonctions*

Le critère d'utilisation systématique et persistante des ressources non partageables du robot permet aussi d'expliciter les cas où une activité peut envoyer directement des requêtes à d'autres modules:

- Si l'activité fille n'utilise aucune ressource non partageable du robot, ce qui signifie qu'elle est compatible avec toutes les autres activités, l'activité mère peut toujours envoyer la requête.
- Si l'activité fille peut créer des conflits potentiels avec d'autres activités, elle doit (comme toute activité) utiliser les ressources dont elle a besoin de façon prévisible et persistante. De plus, il faut compter les ressources allouées à l'activité fille parmi celles allouées à l'activité mère, et l'ensemble doit aussi respecter les règles d'utilisation. Cela implique que:
  - l'envoi de la requête doit être systématique (avoir lieu à chaque exécution); et
  - l'utilisation de la ressource par une (ou des) activité(s) fille(s) doit correspondre approximativement à la durée d'exécution de l'activité mère.

  Comme dans le cas précédent, ces règles peuvent être respectées par subdivision d'un traitement; si des considérations d'ordre pratique empêchent son application intégrale, la conséquence immédiate sera l'augmentation de la probabilité d'interruption de l'activité mère.

Pour illustrer, supposons que les activités `A1`, `A2` et `A3` utilisent une même ressource non partageable du robot (dans le sens qu'elles sont incompatibles entre elles), l'activité `B` n'est incompatible qu'avec une autre instance d'elle-même et `C` ne crée aucun conflit. Nous présentons trois cas qui illustrent les possibilités d'envoi direct de requêtes d'une activité à un module.

Dans le premier cas (fig. 2.10 page suivante), l'exécution de l'activité mère peut évoluer selon deux scénarios différents: soit elle lance `A1` et `C` en exécution parallèle comme activités filles, soit elle lance en séquence et de façon cyclique les activités `A2` et `A3`. Dans ce cas les règles d'envoi direct de requêtes sont respectées: la ressource nécessaire aux activités `A1`, `A2` et `A3` est utilisée de façon prévisible (systématiquement dans tous les scénarios d'exécution possibles de l'activité mère, même si l'activité fille qui l'utilise n'est

pas toujours la même) et persistante (pendant une grande partie du temps d'exécution de l'activité mère). L'activité C n'a pas d'influence sur l'utilisation des ressources.

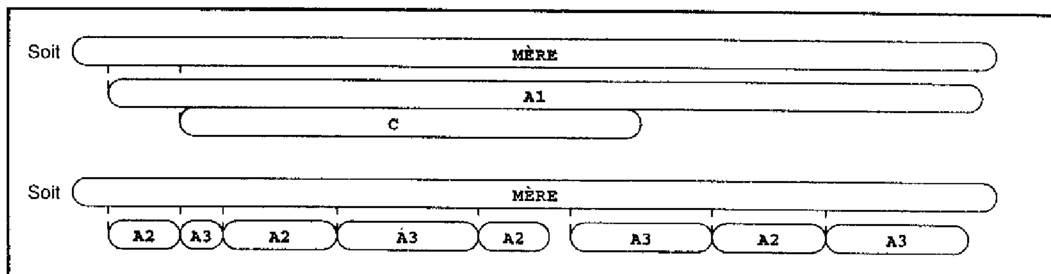


FIG. 2.10 – *Utilisation prévisible et périodique des ressources par les activités filles*

Dans le deuxième cas (fig. 2.11), l'activité mère lance A1 comme activité fille pendant une partie de son exécution, ce qui implique qu'une ressource n'est pas utilisée de façon persistante. Le traitement de l'activité mère peut être réparti entre deux fonctions différentes ou, si cela n'est pas envisageable, la ressource nécessaire à l'activité A1 devra lui être allouée pendant toute son exécution.

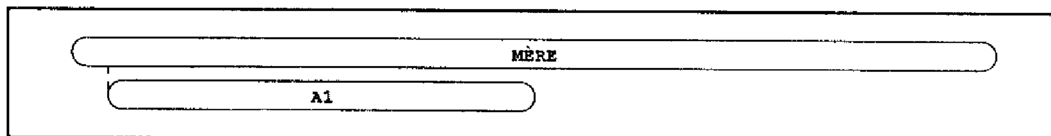


FIG. 2.11 – *Utilisation non persistante d'une ressource par une activité fille*

Dans le troisième cas (fig. 2.12), l'exécution de l'activité mère peut évoluer selon deux scénarios différents: elle lance soit A1 soit B comme activité fille. Dans ce cas les ressources ne sont pas utilisées de façon prévisible: s'il n'est pas désirable de répartir le traitement de l'activité mère entre deux fonctions différentes, il faudra lui allouer à chaque exécution aussi bien les ressources nécessaires à l'activité A1 que celles nécessaires à l'activité B.

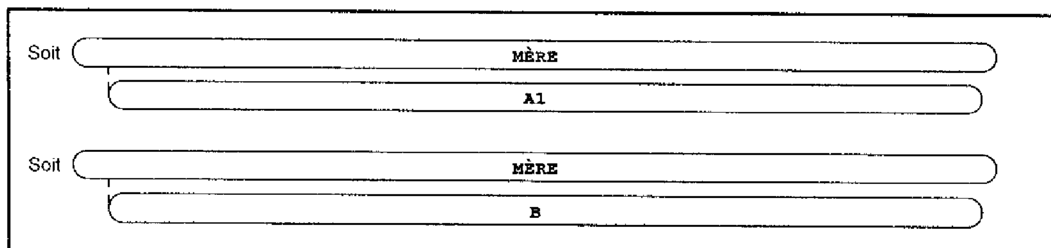


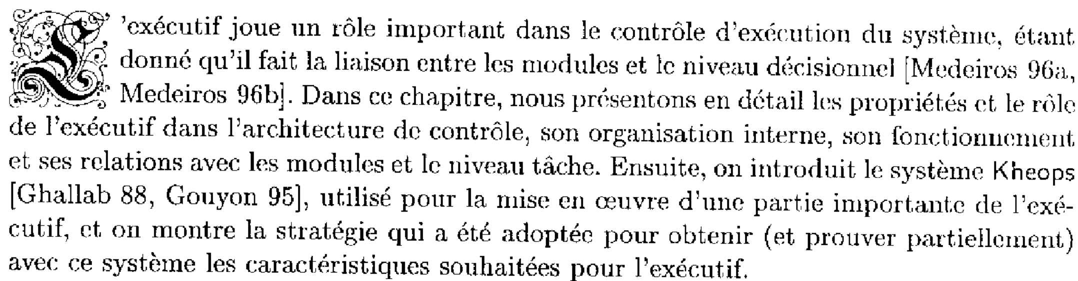
FIG. 2.12 – *Utilisation non prévisible d'une ressource par les activités fille*

2.7 Conclusion

Les modules ne sont pas des entités autonomes, mais font partie d'une architecture plus globale qui doit mener à l'accomplissement de la mission du robot. Comme les contrôleurs des modules ne possèdent qu'une information locale, l'exécutif doit les rassembler pour, entre autres choses, effectuer un contrôle d'exécution qui prenne en compte toutes les activités en cours. Dans le chapitre suivant on présente l'exécutif et ses interaction avec l'ensemble des modules.

Chapitre 3

L'exécutif



L'exécutif joue un rôle important dans le contrôle d'exécution du système, étant donné qu'il fait la liaison entre les modules et le niveau décisionnel [Medeiros 96a, Medeiros 96b]. Dans ce chapitre, nous présentons en détail les propriétés et le rôle de l'exécutif dans l'architecture de contrôle, son organisation interne, son fonctionnement et ses relations avec les modules et le niveau tâche. Ensuite, on introduit le système Kheops [Ghallab 88, Gouyon 95], utilisé pour la mise en œuvre d'une partie importante de l'exécutif, et on montre la stratégie qui a été adoptée pour obtenir (et prouver partiellement) avec ce système les caractéristiques souhaitées pour l'exécutif.

3.1 Propriétés générales

L'exécutif fait la transition entre l'organisation du réseau de modules, basée sur l'établissement dynamique de relations client/serveur, et la structure hiérarchisée du niveau décisionnel. Ses fonctions principales sont:

Déclencher et accompagner l'évolution des activités. L'exécutif, prenant en charge le suivi d'exécution des activités, permet au niveau décisionnel de s'abstraire des détails d'implémentation des fonctions (distribution en modules, protocole de communication, etc.) et le libère des contraintes temporelles fortes qui sont associées à ce rôle. L'exécutif agit donc comme client de tous les modules: du point de vue du niveau décisionnel, tout se passe comme s'il envoyait une requête à l'exécutif et, après un certain temps, l'exécutif lui répondait avec un résultat d'exécution.

Assurer des actions réflexes. Il y a certaines configurations du robot ou de l'environnement, principalement quand la sécurité du système est en jeu, où une réaction bien précise s'impose. Comme le temps de réponse du niveau décisionnel est *a priori* non borné, on peut mettre en œuvre ces réactions au niveau de l'exécutif et profiter de ses bonnes propriétés temporelles (qu'on va présenter par la suite). Les actions accomplies de cette façon ne peuvent être cependant que purement réactives et non réfléchies, parce que l'exécutif n'a pas des capacités de raisonnement et ne connaît pas la tâche globale du robot.

Faire l'interface entre le monde symbolique du niveau décisionnel et la réalité proche des conditions réelles d'exécution des modules. Pour cela, l'exécutif utilise un double dictionnaire de requêtes/répliques, un pour la communication avec le niveau décisionnel, qui représente plutôt les services offerts par le niveau fonctionnel, et un autre pour la communication avec les modules, basé sur les fonctions disponibles. L'exécutif, pour chaque service sollicité, déduit quelle requête il faut envoyer à quel module et avec quels paramètres: pour une requête `GOTO INTERM_GOAL`, par exemple, il faut envoyer au module de pilotage (`PILOT`) la requête `ExecTraj`, avec comme paramètre d'entrée le poster où doit avoir été exportée, par un appel antérieur à une autre fonction, une trajectoire qui mène de la position courante du robot jusqu'à la position du but intermédiaire.

Arbitrer les conflits entre modules. Du fait que normalement un module regroupe les traitements qui utilisent une même ressource du robot, la plupart des conflits entre fonctions implique des fonctions d'un même module, et le contrôleur du module peut gérer lui-même la situation. Dans certains cas, cependant, les fonctions incompatibles résident dans des modules différents, soit parce que deux modules manipulent une même ressource physique du robot, soit à cause d'une incompatibilité logique d'exécution [1]. Il revient alors à l'exécutif d'envoyer une demande explicite d'interruption au module propriétaire de l'activité conflictuelle.

Fournir des informations à propos de l'état des modules. Le niveau décisionnel, pour prendre certaines décisions, peut avoir besoin de connaître, sous un format abstrait, l'utilisation courante des ressources non partageables du robot par les activités des modules. L'exécutif, à partir de sa connaissance du comportement intrinsèque des modules, maintient une description logique de l'état d'utilisation de chacun d'entre eux: cette information lui est d'ailleurs nécessaire pour arbitrer les éventuels conflits entre fonctions.

Dans le contexte d'une mise en œuvre de l'architecture destinée à contrôler un robot réel, il ne suffit pas que l'exécutif remplisse bien ces fonctions. On doit aussi avoir des garanties sur son temps d'exécution: comme tout le flux de contrôle (requêtes et répliques) entre le niveau décisionnel et les modules passe par l'exécutif, ce temps détermine le délai entre la sollicitation d'un service par le niveau décisionnel et le début du traitement dans un module, et entre la fin du service et la prise en compte de ses résultats dans le sens inverse. Le temps de réaction du système global sera donc fortement influencé par la vitesse avec laquelle les informations effectuent cette traversée. Pour que l'exécutif ne devienne pas un goulet d'étranglement de l'architecture de contrôle, il faut le doter de bonnes propriétés temporelles d'exécution en temps réel.

Une première exigence est que le temps d'exécution de l'exécutif soit borné. Cela peut être assuré si on utilise des algorithmes structurellement invariants, dans le sens qu'ils déroulent toujours la même séquence d'instructions (complexité d'ordre 0) avec un temps d'exécution borné par construction. Il faut aussi éviter les appels au système qui risqueraient de bloquer ou ralentir de façon non contrôlable l'exécution (allocation

1. Par exemple, si on décide, par des raisons de sécurité, d'interdire le fonctionnement d'un bras manipulateur monté sur un robot mobile pendant que le robot se déplace.

dynamique de ressources, rendez-vous ou exclusion mutuelle avec des processus extérieurs à l'exécutif, etc.).

De plus, les bornes temporelles doivent être petites par rapport à la dynamique générale du robot, ce qui limite d'une certaine façon la complexité du traitement qu'on peut mettre en place à chaque cycle d'exécution.

Il faut donc concevoir l'exécutif de façon à ce qu'il accomplisse ses tâches dans le respect des contraintes temporelles qui lui sont imposées. La stratégie qui a été adoptée se fonde sur la théorie des automates à états finis (AEFs). Les AEFs ont un pouvoir d'expression suffisant pour représenter le fonctionnement de l'exécutif, sont déterministes et efficaces et peuvent être analysés automatiquement par de nombreux systèmes de vérification. L'exécutif peut donc être vu comme un grand automate où les changements d'état se produisent en raison des signaux externes qui lui parviennent et le franchissement des transitions entre états peut occasionner l'émission de signaux vers l'extérieur (fig. 3.1). Les signaux externes qui peuvent faire évoluer l'automate sont:

- les requêtes du niveau décisionnel; et
- les répliques des modules.

Les signaux que l'exécutif émet lui-même sont:

- les requêtes adressées aux modules; et
- les répliques envoyées au niveau décisionnel.

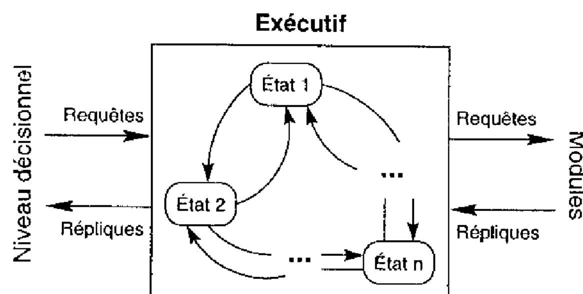


FIG. 3.1 – *L'exécutif vu comme un automate à états finis*

Étant donné que chaque cycle de l'exécutif a un temps d'exécution réduit, on peut considérer comme négligeable l'évolution du système pendant son fonctionnement et donc considérer comme satisfaite l'hypothèse synchrone[2]. Cela n'est possible que parce que l'exécution proprement dite des actions du robot est faite au niveau des modules, l'exécutif ne réagissant qu'aux transitions marquant les étapes clefs des activités (activation, début et fin d'exécution, interruption, etc.).

L'adoption de l'hypothèse synchrone a permis l'utilisation du système Kheops pour le développement d'une partie de l'exécutif, avec les facilités de mise en œuvre et les possibilités de vérification formelle qu'il apporte. Kheops a été choisi (entre les langages synchrones)

2. La sortie est synchrone avec l'entrée, les actions internes sont instantanées et les communications sont faites par transmission instantanée [Benveniste 91].

à cause de la facilité de programmation avec des règles, qui permettent une traduction directe du fonctionnement des modules et l'adjonction incrémentale de nouvelles règles selon l'application ou l'évolution du système.

Le prix à payer pour ce modèle "synchronisé" de l'exécutif est qu'il n'y a aucune liaison directe entre lui et l'environnement. Il ne prend en compte les changements dans l'état du monde et du robot qu'à travers les répliques qui lui arrivent des modules. Il revient au niveau décisionnel, ou à l'exécutif lui-même, d'activer les surveillances nécessaires pour que les événements qui peuvent avoir une influence sur le contrôle d'exécution soient détectés.

3.2 La communication avec les modules

L'exécutif utilise le protocole présenté dans la section 2.5 pour sa communication avec les modules. Il peut envoyer des requêtes exécutables[3] (qu'on va nommer `RQST EXEC`) ou des requêtes d'interruption (`RQST ABORT`) et recevoir des répliques intermédiaires (`REPLY INTERM`) ou finales. Pour ce qui concerne les répliques finales, on les classe en catégories:

`REPLY END` – indique une fin correcte de l'activité ou une terminaison à cause d'une erreur intrinsèque. Correspond aux répliques finales du type `END` (tableau 2.1 page 40).

`REPLY INT` – signale l'interruption de l'activité (répliques finales du type `INTERRUPTED`).

`REPLY ERROR` – toutes les fins d'exécution causées par des situations exceptionnelles, extérieures à l'activité (types `CS_ERROR`, `TIMEOUT`, `FINAL_ERROR` ou `SERVER_ERROR`).

`REPLY ZOMBIE` – répliques des activités qui sont figées dans un module (type `ZOMBIE`).

`REPLY ABORT` – réplique d'une demande d'interruption. Une activité est interrompue au moyen d'une requête spécifique (*mod*`AbortRqstSend`). Ainsi, après l'interruption, le client recevra deux répliques: la première est la réplique finale de l'activité (du type `INTERRUPTED`, et donc classée comme `REPLY INT`) et la seconde signale la fin normale de la procédure d'interruption. Cette deuxième réplique, du type `END`, sera classée comme `REPLY ABORT` par rapport à l'activité interrompue.

Généralement, `REPLY INT` et `REPLY ABORT` sont redondantes et arrivent presque simultanément, ce qui fait que `REPLY ABORT` n'a pas beaucoup d'importance. Cependant, quand une activité est en état `ZOMBIE`, elle a déjà envoyé sa réplique finale: à ce moment `REPLY ABORT` devient importante parce qu'elle sera la seule réplique reçue après la demande d'interruption.

3.3 L'architecture et le fonctionnement

On représente dans la figure 3.2 page ci-contre l'organisation interne de l'exécutif. Il est composé de cinq éléments principaux:

Liste de description des services: cette liste statique (non modifiable en exécution) contient une description de chaque service que le niveau décisionnel peut solliciter

3. Par rapport aux modules, les requêtes exécutables peuvent être des requêtes d'exécution ou des requêtes de contrôle.

du niveau fonctionnel. Elle correspond à un dictionnaire des requêtes reconnues par l'exécutif. Pour chaque service, la liste décrit la fonction qu'il faut activer, au près de quel module, les paramètres à fournir, etc.

Liste d'activités: stocke des informations (principalement l'état d'exécution) sur les activités qui ont été démarrées par l'exécutif et qui sont en cours dans les divers modules.

Gestionnaire de requêtes: surveille l'arrivée des requêtes du niveau décisionnel, fait évoluer la liste d'activités, sollicite la prise en compte de la requête par le contrôleur de ressources et, si autorisé, envoie la requête correspondante vers les modules.

Gestionnaire de répliques: surveille l'arrivée des répliques des modules, fait évoluer la liste d'activités, sollicite la prise en compte de la réplique par le contrôleur de ressources et envoie la réplique appropriée au niveau décisionnel.

Contrôleur de ressources: représente d'une manière globale et abstraite l'état d'utilisation des ressources du robot par les modules et décide de l'activation et interruption des fonctions selon les éventuels conflits crées.

Niveau décisionnel
Liste de description des services
Liste d'activités
Gestionnaire de requêtes
Gestionnaire de répliques
Contrôleur de ressources
Modules

FIG. 3.2 – *L'architecture de l'exécutif*

Avant de décrire avec plus de détails le contenu et le fonctionnement de ces éléments, nous donnons un aperçu du fonctionnement nominal de l'exécutif.

Quand une requête d'exécution arrive du niveau décisionnel, le gestionnaire de requêtes exécute un cycle. Initialement, la liste de description des services est examinée pour identifier la requête. Si elle est reconnue, une nouvelle activité est incluse dans la liste d'activités et son état d'exécution est marqué `WAIT`. Cet état, lié au fonctionnement de l'exécutif, ne correspond à aucun des états du graphe de contrôle d'une activité au niveau des modules (fig. 2.3 page 33): on pourrait virtuellement le situer entre les états `ETHER` et `INIT`. Ensuite, le contrôleur de ressources est appelé pour décider du démarrage de l'activité et, le cas échéant, le prendre en compte dans sa représentation de l'état d'utilisation des ressources.

Si le contrôleur de ressources autorise l'exécution, le gestionnaire de requêtes fait passer l'état d'exécution de l'activité vers `INIT`[4] et envoie la requête appropriée au module concerné (selon les informations de la liste de description des services). Cela conclut un cycle d'exécution typique du gestionnaire de requêtes: un nouveau cycle aura lieu quand une nouvelle requête du niveau décisionnel arrivera.

L'activité qu'on vient d'initier, après un certain temps et si on suppose qu'il s'agit d'une requête d'exécution, envoie une réplique intermédiaire, suivie d'une réplique finale à la fin du traitement. Chaque réplique qui arrive des modules occasionne l'exécution d'un cycle du gestionnaire de répliques. Dans le cas de la réplique intermédiaire, l'état de l'activité (dans la liste d'activités) est mis à jour et passe à `EXEC`. Pour la réplique finale, le gestionnaire de répliques envoie au niveau décisionnel une réplique finale avec le bilan d'exécution et l'exclut de la liste d'activités (ce qui, du point de vue conceptuel, équivaut à faire revenir l'activité à l'état `ETHER`). Le contrôleur de ressources est informé de l'arrivée de chaque réplique et actualise son information à propos de l'état d'utilisation des ressources.

3.3.1 La liste de description des services

La liste de description des services joue le rôle d'un dictionnaire de conversion entre les requêtes de services connues par le niveau décisionnel et les requêtes fonctionnelles correspondantes au niveau des modules. Pour chaque service la liste contient:

1. le nom du service;
2. la requête fonctionnelle correspondante, avec son module d'appartenance et s'il s'agit d'une requête de contrôle ou d'exécution;
3. les éventuels paramètres d'entrée à être transmis au module avec la requête et une fonction optionnelle pour composer ces paramètres à partir de données existantes;
4. les éventuels paramètres de sortie à être transmis au module avec la requête et une fonction optionnelle qui traite ces données au cas où des résultats doivent être envoyés au niveau décisionnel; et
5. les temps maximaux d'attente associés à ce service: autorisation de démarrage à l'intérieur de l'exécutif, arrivé de la réplique intermédiaire (s'il s'agit d'une requête d'exécution) et arrivée de la réplique finale.

Plusieurs services peuvent faire référence à la même requête fonctionnelle: il suffit qu'un changement de paramètres modifie, du point de vue du niveau décisionnel, le service offert par l'appel à cette fonction. Pour illustrer, prenons le cas d'une fonction `planCalcTraj`, d'un module de planification de trajectoire, qui à partir d'une représentation de l'environnement calcule un parcours exécutable par le robot de la position courante jusqu'à une position donnée. Cette fonction peut fournir plusieurs services: `CALC_TRAJ_EXIT`, quand on lui passe comme paramètre d'entrée la position de la sortie, `CALC_TRAJ_GOAL`, quand il s'agit de la position du but final de la mission, et ainsi de suite.

4. La représentation complète des états des activités au niveau de l'exécutif sera présentée ultérieurement (section 3.3.2); mais, d'une manière générale, les états ont la même signification que leurs équivalents au niveau des modules (fig. 2.3 page 33).

Une fonction d'un module peut aussi correspondre à plusieurs services quand il y a des conflits “logiques” avec d'autres fonctions (on reviendra à cette possibilité dans la section 3.6). Quand deux activités ne peuvent pas coexister parce qu'elles utilisent la même ressource physique non partageable du robot (un moteur, par exemple), le conflit est absolu et existera toujours. Il y a d'autres situations, cependant, où l'existence de conflit entre activités (normalement dans des modules différents) est relative et dépend du contexte d'exécution. Si l'on reprend l'exemple d'un robot mobile possédant un bras manipulateur, l'utilisation du bras peut être incompatible avec un déplacement du robot si on veut saisir un objet mais pas si on veut déposer l'objet saisi sur le robot. L'utilisation du bras peut donc se faire par deux services différents, dont un seul est incompatible avec un déplacement simultané du robot, mais qui exécutent la même fonction au niveau du module. Le niveau décisionnel choisit quel service utiliser selon sa connaissance globale de la tâche du robot et du contexte d'exécution.

D'un autre côté, certaines fonctions des modules peuvent n'offrir aucun service au niveau décisionnel et donc ne pas être représentées dans la liste de description des services. Il s'agit normalement de fonctions de bas niveau, qui ne sont activées que par d'autres fonctions dans le cadre d'une relation client/serveur. L'exécutif connaît l'existence de ces fonctions, et les prend en compte implicitement quand il lance l'activité mère, mais ne les rend pas directement accessibles au niveau décisionnel. Des exemples typiques sont les fonctions d'asservissement des moteurs, qui normalement ne sont utilisées que comme serveurs du module de génération de trajectoires.

3.3.2 La liste d'activités

La liste d'activités contient, pour chacune des activités qui auront été lancées par l'exécutif[5]:

1. le service qui a déclenché l'activité (comme on l'a vu, cette correspondance peut ne pas être unique).
2. deux identificateurs: un qui identifie l'activité par rapport au module auquel elle appartient et l'autre qui désigne le service par rapport au niveau décisionnel. Tous les envois de requêtes et de répliques utilisent un de ces deux identificateurs.
3. une représentation de son état d'exécution. Les états possibles, avec les transitions entre eux, forment un graphe qu'on dénomme graphe de représentation d'une activité (fig. 3.3 page suivante). Le graphe de représentation d'une activité présente plusieurs similitudes avec le graphe de contrôle au niveau des modules (fig. 2.3 page 33), dont il essaye de reconstituer le fonctionnement.
4. l'instant d'inclusion dans la liste (pour vérifier le temps maximal d'attente).

En plus de l'état fictif `ETHER`, qui symbolise le fait que l'activité n'existe pas dans la liste d'activités, une activité peut être dans un état `WAIT`, qui indique qu'elle attend une autorisation du contrôleur de ressources pour démarrer, ou dans un des états `INIT`, `EXEC`,

5. Les activités filles, comme on a vu dans la section 1.6.4, sont gérées directement par l'activité mère.

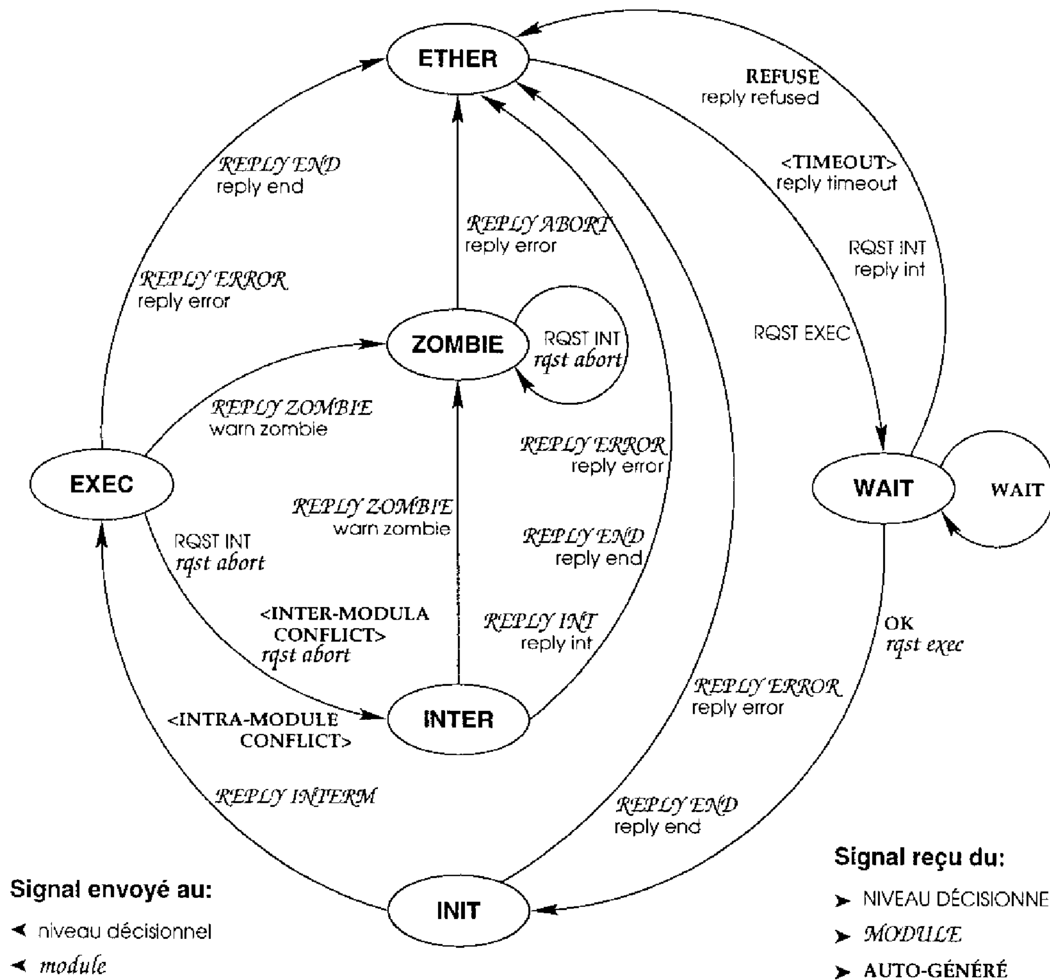


FIG. 3.3 – *Le graphe de représentation des activités*

INTER ou ZOMBIE: dans ces cas, à la connaissance de l'exécutif, l'activité en question est dans l'état du même nom dans son graphe de contrôle au niveau des modules.

Les signaux qui font évoluer l'état du graphe de représentation d'une activité viennent:

- de l'exécutif lui-même (du contrôleur de ressources);
- du niveau décisionnel: des requêtes pour exécuter (RQST EXEC) ou interrompre (RQST INT) un service; et
- des modules: des répliques intermédiaires (REPLY INTERM) ou des répliques finales qui rendent compte de la fin nominale (REPLY END) ou anormale (REPLY ERROR, REPLY ZOMBIE) de l'exécution d'une activité ou de son interruption (REPLY INT, REPLY ABORT);

Les changements d'état dans le graphe de représentation sont effectués par les divers composants de l'exécutif, et seront décrits par la suite. Les transitions peuvent occasionner l'envoi de signaux vers les modules (requêtes) ou vers le niveau décisionnel (avertissements ou répliques).

3.3.3 Le gestionnaire de requêtes

Le gestionnaire de requêtes est activé à chaque fois qu'une requête arrive du niveau décisionnel. Initialement, il parcourt la liste de description de services pour identifier la requête: si le service n'est pas reconnu, un message d'erreur est immédiatement envoyé au client [6]. Si ce service possède une fonction d'entrée, elle est appelée pour mettre les paramètres d'entrée dans le format approprié. L'activité correspondant au service est ensuite incluse dans la liste d'activités, avec un état d'exécution WAIT, et le gestionnaire de requêtes peut alors actionner le contrôleur de ressources pour qu'il autorise son démarrage. La requête provenant du niveau décisionnel contient un identificateur numérique unique: tout contact ultérieur avec le client sera fait par des requêtes et répliques qui contiendront cet identificateur.

Si un service dans un cycle précédent d'exécution n'a pas obtenu l'autorisation pour démarrer son activité et qu'il est resté dans un état d'attente (WAIT), le gestionnaire de requêtes redemande l'autorisation à chaque nouveau cycle. Si elle est accordée, le traitement se poursuit comme s'il s'agissait d'un service qui vient d'être demandé. Dans le cas contraire, et si le temps maximal autorisé pour le démarrage du service ne s'est pas écoulé, il reste en attente. Il peut se produire qu'entre-temps le client ne soit plus intéressé par son exécution, et le signale par l'envoi d'une requête d'interruption (RQST INT): dans ce cas, le gestionnaire de requêtes envoie au niveau décisionnel une réplique qui confirme l'interruption (REPLY INT) et élimine l'activité de la liste d'activités.

Un temps d'attente est associé à chaque service, qui indique le délai maximal autorisé avant que l'activité correspondante se mette en exécution: quand il est dépassé, une réplique d'erreur (REPLY TIMEOUT) est envoyée au client et l'activité est exclue de la liste d'activités.

Quand l'autorisation de démarrage d'une activité est sollicitée au contrôleur de ressources, plusieurs réponses sont possibles:

OK – le démarrage de l'activité est autorisé.
WAIT – l'activité doit attendre.
REFUSE – l'exécution de l'activité est définitivement refusée: le gestionnaire de requêtes signale l'événement au client par l'envoi d'une réplique au niveau décisionnel (REPLY REFUSED) et l'élimine de la liste d'activités.

En cas d'autorisation d'exécution, le gestionnaire de requêtes change l'état de l'activité en INIT et envoie ensuite la requête appropriée (décrite dans la liste de description des services) à un des modules. Cela conclut son cycle d'exécution.

6. Excepté pendant les phases de mise en œuvre du système, ce genre d'erreur ne doit jamais se produire pendant le fonctionnement normal du système.

3.3.4 Le gestionnaire de répliques

Le gestionnaire de répliques est activé quand une réplique arrive des modules. Initialement, le contrôleur de ressources est informé de l'événement. Ensuite, s'il s'agit d'une réplique intermédiaire, le gestionnaire fait passer l'état de l'activité correspondante de INIT vers EXEC. Pour les répliques finales, plusieurs possibilités existent, selon l'état de l'activité et le type de réplique:

- Les activités déclenchées par des requêtes de contrôle[7] ne reçoivent pas de réplique intermédiaire et sont donc dans l'état INIT quand la réplique finale arrive. Dans ce cas, le gestionnaire de répliques envoie une réplique finale (REPLY END) au niveau décisionnel et élimine l'activité correspondante de la liste d'activités. Le client ne verra donc pas la différence entre les services qui, au niveau des modules, correspondent à des requêtes d'exécution et ceux fournis par des requêtes de contrôle.
- Quand l'activité est en exécution (état EXEC), le comportement du gestionnaire de répliques dépend du type de réplique:
  - En cas d'exécution normale ou avec erreur, la réplique finale appropriée (REPLY END ou REPLY ERROR) est envoyée au niveau décisionnel et l'activité peut sortir de la liste d'activités.
  - Si la réplique finale signale que l'activité correspondante est restée figée au niveau du module dans un état ZOMBIE, le gestionnaire de répliques adopte la même procédure au niveau de l'exécutif: il change l'état de l'activité dans la liste d'activités, et envoie un avertissement spécifique (WARN ZOMBIE) au client pour signaler l'événement.

Certains services doivent envoyer des résultats (et non seulement le bilan) d'exécution au niveau décisionnel. La description de ces services dans la liste de description de services inclut une fonction de sortie, qui se charge de mettre dans le format approprié et d'envoyer ce résultat. Cette fonction est appelée par le gestionnaire de répliques juste après l'envoi de la réplique finale.

3.3.5 Le contrôleur de ressources

Le contrôleur de ressources gère les conflits entre services. Il peut être vu comme un automate à états finis, où à chaque état on associe une description de l'utilisation de chacune des ressources non partageables du robot. À chaque transition entre états il y a des signaux associés que le contrôleur de ressources doit émettre. Les signaux sont destinés au composant de l'exécutif qui a déclenché le cycle du contrôleur de ressources et dépendent du type d'événement signalé:

- Quand le contrôleur de ressources reçoit une demande de démarrage d'une activité, il peut répondre de plusieurs façons:

  OK - le démarrage de l'activité est autorisé.

7. Au niveau des modules, on ne parle d'activités que pour les requêtes d'exécution, étant donné que les requêtes de contrôle sont traitées de façon considérée instantanée. L'exécutif ne fait pas cette distinction, et considère que chaque demande de service crée une activité.

REFUSE - l'activité ne peut pas être exécuté. Cela arrive fondamentalement quand il y a des activités figées en état ZOMBIE.

WAIT - il est impossible de commencer l'exécution en ce moment mais la demande peut être resoumise plus tard.

ERROR - service inconnu (ne devrait arriver qu'en phase de développement).

- Si le contrôleur est informé de l'arrivée d'une requête d'interruption de service ou d'une réplique intermédiaire ou finale, les réponses possibles sont:

  OK - l'événement a été pris en compte.

  ERROR - à la connaissance du contrôleur de ressources, le service auquel l'événement fait référence n'est pas (ou ne doit pas être) en exécution.

Le contrôleur de ressources est le seul qui puisse décider de toutes les actions qui peuvent occasionner l'interruption d'une activité, ce qui peut se produire dans trois cas:

1. Le niveau décisionnel envoie une demande explicite d'interruption de l'activité, à laquelle le contrôleur réagit par une autorisation d'envoi de la requête d'interruption au module concerné.
2. Une activité conflictuelle dans le même module veut être démarrée; si le contrôleur l'autorise, l'activité initiale sera interrompue sans qu'il ait besoin de prendre aucune mesure spécifique.
3. On veut démarrer une activité conflictuelle dans un autre module; pour cela, il faut que préalablement une demande d'interruption soit envoyée à l'activité en exécution. Le contrôleur ajoute cette demande d'interruption dans la liste d'activités, et elle sera traitée par le gestionnaire de requêtes comme celles venues du niveau décisionnel.

Quand le contrôleur de ressources sait qu'une activité sera interrompue (soit parce qu'une requête d'interruption explicite va être envoyée, soit parce qu'une activité conflictuelle va être autorisée), il agit sur la liste de description d'activités pour changer son état d'exécution en INTER. Dans des conditions normales de fonctionnement du robot, aucune réplique qui signale une interruption ne peut arriver sans que l'activité correspondante soit dans l'état INTER du graphe de représentation d'activités (fig. 3.3 page 52).

3.4 Gestion des situations d'erreur

Il y a plusieurs situations d'exécution non nominale que l'exécutif doit gérer. Pour des cas spécifiques, principalement quand la sécurité du robot est en jeu, le concepteur peut prévoir des réactions appropriées (on mentionnera les réactions réflexes à la section 3.5). Pour les autres, l'exécutif prévoit un traitement standardisé.

3.4.1 Les erreurs signalées par les modules

La réaction aux erreurs intrinsèques dépend du contexte et de la tâche globale du robot, échappant aux compétences de l'exécutif. Par exemple, la non reconnaissance d'un objet dans une image peut être due à une défaillance de la caméra mais peut aussi tout

simplement indiquer qu'on doit la réorienter: il n'y a que le niveau décisionnel, avec sa connaissance globale de la situation et de la tâche, qui peut décider à ce propos. Du point de vue du contrôle de l'exécution, ce genre d'activité s'est terminée normalement, étant donné qu'elle est allée jusqu'au but de ses possibilités algorithmiques. L'exécutif va traiter cet événement comme une fin d'exécution normale: le niveau décisionnel recevra une réplique de fin nominale d'exécution (`REPLY END`) avec le bilan d'exécution correspondant. Il revient au concepteur de la fonction du module de prévoir des bilans suffisamment explicites pour permettre au niveau décisionnel de déterminer la suite des actions.

Pour les erreurs extrinsèques, l'exécutif signale au niveau décisionnel l'échec complet du service. Si l'activité a décidé que l'erreur est assez grave pour passer en `ZOMBIE`, l'exécutif respecte cette décision. Mais l'exécutif peut décider de figer une activité finie avec erreur même si le module concerné ne l'a pas fait. Cette différence de comportement peut exister en raison du fait que la politique de jugement de la gravité des erreurs est propre à chaque module (même à chaque fonction), alors que l'exécutif peut vouloir mettre en place une politique globale d'exigence systématique d'une réaction du niveau décisionnel face à certaines catégories d'erreurs.

Chaque fois qu'une activité se finit en `ZOMBIE`, l'exécutif signale l'événement au niveau décisionnel, en écrivant un message spécifique (`WARN ZOMBIE`) dans sa base de données, et refuse les nouvelles requêtes pour ce module. Il s'agit là de la seule exception au paradigme de la réactivité du niveau fonctionnel (toute nouvelle requête est considérée prioritaire par rapport aux précédentes et acceptée): on a adopté ce comportement (aussi bien au niveau des modules qu'au niveau de l'exécutif) pour exiger des niveaux supérieurs qu'ils signalent explicitement qu'ils ont pris connaissance des erreurs graves.

Le niveau décisionnel doit réagir à un `ZOMBIE` par l'envoi d'une requête d'interruption de l'activité, qui sera traitée selon deux possibilités:

- si au niveau du module l'activité n'est pas dans un état `ZOMBIE`, le module est débloqué (le contrôleur de ressources recommence à accepter les requêtes) et la réplique finale de l'activité (`REPLY ERROR`) est envoyée immédiatement au niveau décisionnel.
- si l'activité est figée dans le module, l'exécutif envoie une requête d'interruption. Quand la réplique à cette requête (`REPLY ABORT`) arrivera, le module sera débloqué dans le contrôleur de ressources et la réplique finale `REPLY ERROR` de l'activité en question sera envoyée au niveau décisionnel.

Par rapport à la communication avec les modules, ce protocole de communication avec le niveau décisionnel présente deux différences: il n'y a pas de répliques pour une requête d'interruption et la réplique finale d'une activité en état `ZOMBIE` n'est envoyée qu'après son interruption (la réplique finale signale une erreur, et non une interruption).

3.4.2 Les erreurs détectées par l'exécutif

Il y a une longue série de situations qui ne doivent jamais se produire dans un fonctionnement normal du robot. On peut citer à titre d'exemple:

- Une réplique d'interruption qui arrive des modules concernant une activité qui n'est pas dans un état `INTER` dans son graphe de représentation.

- Une réplique quelconque d'une activité non présente dans la liste d'activités ou qui n'est pas en exécution à la connaissance du contrôleur de ressources.

Toutes ces situations indiquent des dysfonctionnements graves du système de contrôle, et sont normalement causées par des erreurs dans la phase de développement. Dans ce cas, l'exécutif se met lui-même dans un état d'erreur général (`ERROR`) et refuse toute requête pour n'importe quel module. Cet état est prévu simplement comme une étape de terminaison contrôlée de toutes les activités en cours pour permettre une révision du système.

3.5 Les réactions réflexes

Les réactions qu'on peut mettre en œuvre au niveau de l'exécutif doivent être entièrement réflexes (et donc applicables en toute situation), étant donné que la tâche globale du robot n'est pas connue à ce niveau. Elles doivent aussi être ponctuelles (normalement, seulement l'envoi d'une requête), pour ne pas compromettre les performances temporelles de l'exécutif.

Les réactions sont très dépendantes du système. Mais les deux domaines où on peut envisager le plus facilement son apparition sont les situations d'urgence et la reprise d'erreurs. La réaction la plus courante (et la plus simple) consiste à avertir l'opérateur de la situation. Le composant de l'exécutif chargé de mettre en place les réactions dépend du facteur qui la déclenche:

- Les réactions à des répliques des modules seront mises en place dans le gestionnaire de répliques. Il peut s'agir d'une réaction à une réplique particulière (si une surveillance de danger retourne une réplique positive, envoyer une requête d'arrêt immédiat du robot) ou à toute une catégorie de répliques (en cas de défaillance du système, lancer une activité de test).
- Les réactions à des configurations d'activité seront effectuées par le contrôleur de ressources (comme retarder le démarrage d'une nouvelle activité quand deux autres grandes consommatrices de mémoire sont déjà en activité).

3.6 Gestion des conflits

Si on excepte l'état `ZOMBIE`, toutes les requêtes qui arrivent à l'exécutif sont acceptées. Mais cette règle ne répond pas à toutes les questions qu'on peut se poser avant de démarrer une activité:

- à part les activités qui utilisent une même ressource du robot, et qui seront obligatoirement interrompues par les modules, est-ce qu'il faut interrompre d'autres activités avant de la démarrer?
- pour les activités où il y a un temps d'attente autorisé avant de commencer l'exécution, est-ce qu'il faut interrompre tout de suite les éventuelles activités en conflit ou attendre qu'elles se terminent?

L'exécutif ne peut pas prendre ces décisions de façon autonome parce qu'elles exigent des connaissances globales à propos du contexte d'exécution et de la tâche du robot qui le dépassent. Mais il peut proposer au niveau décisionnel plusieurs services, chacun d'entre eux se comportant différemment par rapport à d'éventuelles autres activités conflictuelles en exécution. Si l'on revient à l'exemple canonique de l'utilisation d'un bras manipulateur et ses relations avec un déplacement simultané du robot mobile sur lequel il est monté, on peut envisager l'existence de trois services:

1. `ArmGoto` – le mouvement du bras peut être réalisé indépendemment d'un éventuel déplacement du robot, avec lequel il ne crée pas de conflit.
2. `ArmGotoWaitStop` – si le robot n'est pas immobilisé, le mouvement du bras doit attendre la fin du déplacement.
3. `ArmGotoStop` – le mouvement du bras doit commencer au plus tôt, avec une interruption préalable d'éventuels déplacements du robot.

Il revient au contrôleur de ressources de mettre en œuvre cette politique de résolution des conflits "logiques". Pour cela, il faut qu'il raisonne non seulement en termes d'activités, mais aussi des services avec lesquels ces activités sont associées. Chaque fois qu'il doit démarrer ou interrompre une activité, il prend en compte:

- la politique de priorités associée au service; et
- les incompatibilités "physiques" entre l'activité associée au service et les autres activités du même module.

Il est clair qu'on pourrait laisser la gestion de ces problèmes à la charge du niveau décisionnel: comme il connaît les services qu'il a demandé, il peut les interrompre si nécessaire avant de demander un nouveau service. Mais cette solution alourdit sans nécessité le travail du niveau décisionnel, augmente le volume de requêtes de contrôle adressées à l'exécutif et ralentit le système, compte tenu du temps de réaction plus petit de l'exécutif. En plus, elle suppose une connaissance complète et permanente des activités en cours par le niveau décisionnel, ce qui est délicat à mettre en œuvre étant donné sa structuration distribuée.

3.6.1 La représentation de l'information dans le contrôleur de ressources

Si l'on considère le contrôleur de ressources comme un automate à états finis, où à chaque état on associe une utilisation différente des ressources du robot, il faut prévoir un nombre d'états suffisant pour gérer tous les conflits potentiels entre services, mais en assurant un temps de traitement et une quantité d'informations stockées compatibles avec les contraintes temporelles et matérielles de l'exécutif. On présente par la suite différentes approches possibles pour concevoir cet automate, avec les raisons qui ont conduit à l'adoption du modèle finalement choisi.

Une des représentations qui vient la première à l'esprit consiste à associer à chaque état une liste de services, avec autant d'éléments que de ressources non partageables du

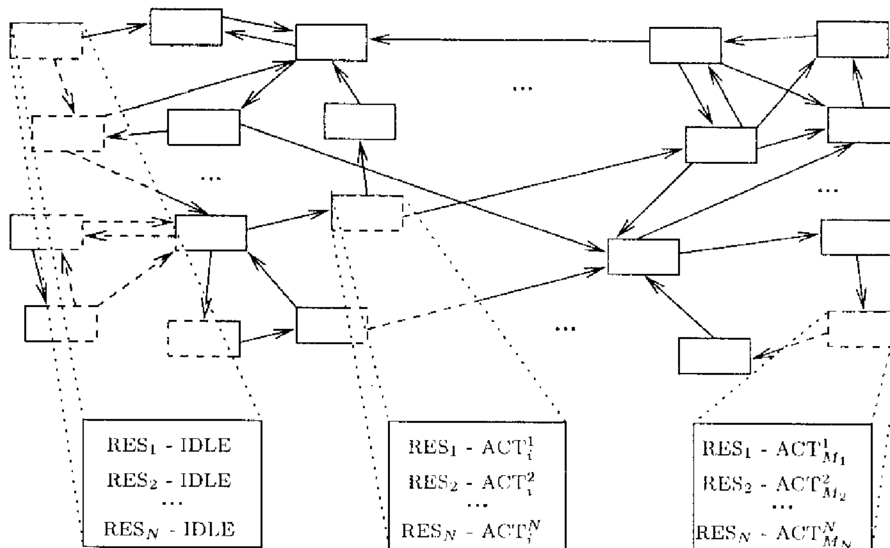


FIG. 3.4 – *Une conception du contrôleur de ressources basée sur les ressources*

robot. Chaque élément de cette liste ordonnée contient le service qui utilise la ressource correspondante, ou IDLE si la ressource est libre (fig. 3.4).

Cette représentation est minimale en nombre d'états, mais ne peut pas être utilisée pour deux raisons:

1. On ne connaît pas nécessairement la liste exhaustive des ressources non partageables du robot. Dans la description formelle des modules (section 2.4), on indique simplement les incompatibilités entre activités (dans le sens que l'exécution de l'une présuppose l'interruption des autres), sans en expliciter la cause.
2. Ce modèle présuppose que tout service qui utilise une ressource est automatiquement incompatible avec tous les autres qui le font aussi. Mais on peut avoir des incompatibilités non réciproques (A exige l'interruption de B pour s'exécuter, mais pas le contraire), ce qui n'est pas représentable avec cette méthode.

Une deuxième possibilité consiste à associer aux états une liste plus grande, avec autant d'éléments que de services existants, où la valeur IDLE ou EXEC d'un élément indique si le service correspondant est en exécution. Si cette approche contient toute l'information nécessaire au fonctionnement du contrôleur de ressources, elle génère des états qui ne sont pas utiles: on n'aura jamais une situation où deux services mutuellement incompatibles seront en exécution au même temps.

La représentation que nous avons retenue est très proche de la première. Comme seule modification, on ne parle plus de groupes de services qui utilisent une même ressource, mais de groupes de services qui maintiennent des relations d'incompatibilité entre eux. On associe à chaque état une liste de services, où les éléments de cette liste ordonnée indiquent quel service (ou IDLE si aucun) du groupe correspondant est en exécution. Cela permet de

ne pas expliciter la cause exacte de l'incompatibilité et d'avoir des relations plus complexes qu'une simple incompatibilité totale et réciproque à l'intérieur d'un groupe de services.

La question majeure est la détermination du nombre minimal de groupes de services qui peut contenir l'information nécessaire à la résolution des situations possibles de conflit. Pour cela, on représente les incompatibilités dans un graphe orienté $G = (X, U)$, comme celui de la figure 3.5, où les sommets (X) sont les services et les arcs (U) indiquent les incompatibilités: un arc de `A` vers `B` signifie que, pour exécuter `A`, il faut d'abord interrompre `B`. Il s'agit d'un graphe simple (sans arcs en parallèle dans le même sens) qui peut contenir des boucles pour les services qui sont incompatibles avec une autre instance d'eux-mêmes.

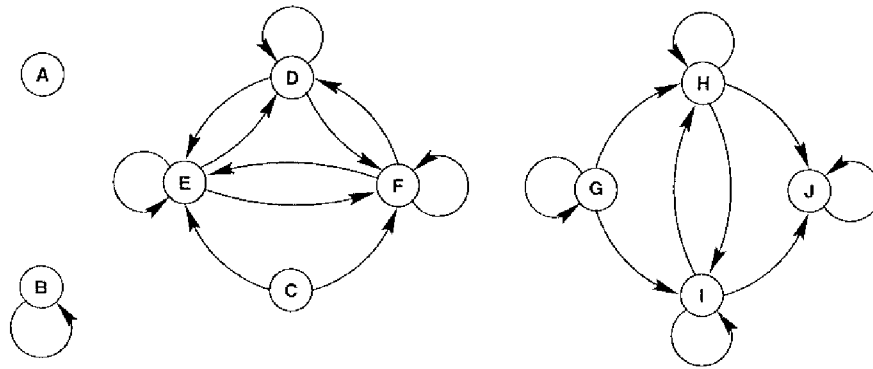


FIG. 3.5 – *Un exemple de graphe d'incompatibilités entre services*

Dans le graphe d'incompatibilités, il faut d'abord localiser les sous-graphes complets et symétriques. Les services correspondants sont entièrement incompatibles, et il suffit de garder l'information précisant lequel d'entre eux est en exécution pour résoudre tous les conflits éventuels qui les concernent. On peut remplacer tous ces sommets et les arcs entre eux par un macro-sommet avec une boucle, et obtenir un graphe réduit (fig. 3.6). L'élément correspondant au macro-sommet dans la liste de services qui décrit l'état de l'automate pourra assumer autant de valeurs que de sommets compactés, plus `IDLE` si aucun d'entre eux est en exécution.

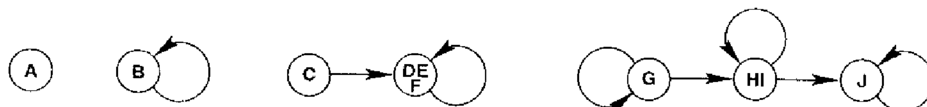


FIG. 3.6 – *Le graphe d'incompatibilités simplifié*

Avec le graphe réduit il devient plus simple de déterminer le nombre de groupes de services qu'il faut associer à chaque état de l'automate du contrôleur de ressources. L'analyse se fait dans chaque sous-graphe connexe: on doit créer autant de groupes de services que le nombre de services interruptibles qui peuvent s'exécuter simultanément. Dans le graphe réduit, cela peut être déterminé, dans la plupart des cas, par une règle très simple: il faut autant de groupes de service que de sommets qui sont extrémité terminale d'au

moins un arc. Pour l'exemple de la figure 3.6 page précédente il y a quatre sous-graphes non connexes:

1. Comme l'activité `A` n'est jamais interrompue, il n'est pas nécessaire de garder des information sur ce sous-graphe.
2. Pour le deuxième sous-graphe, il faut créer un groupe de services. L'élément correspondant dans la liste qui décrit l'état de l'automate aura les valeurs `B` ou `IDLE`. Il faut garder cette information pour que le contrôleur de ressources sache, quand une requête d'activation du service `B` arrive, s'il y a ou non une instance de ce service qui s'exécute et qui sera interrompue par la nouvelle requête.
3. Le troisième sous-graphe requiert la création d'un seul groupe d'activités (`D`, `E`, `F` ou `IDLE`), étant donné que `C` n'est jamais interrompue.
4. Trois groupes d'activités, (`G` ou `IDLE`), (`H`, `I` ou `IDLE`) et (`J` ou `IDLE`), sont nécessaires pour représenter le quatrième sous-graphe.

Le nombre d'états de l'automate qui représente le contrôleur de ressources est égal au nombre de combinaisons possibles entre toutes les valeurs envisageables pour les éléments qui représentent les différents groupes de services. Si le groupe numéro i est composé de M_i services, le nombre d'états est de $\prod_{i=1}^{N}(M_i + 1)$, où N est le nombre de groupes, ce qui peut être assez conséquent. Pour notre petit exemple, N=5, M_1=1 (`B`), M_2=3 (`DEF`), M_3=1 (`G`), M_4=2 (`HI`) et M_5=1 (`J`), ce qui fait que l'automate aura 96 ($2\times4\times2\times3\times2$) états (plus quelques uns pour représenter les situations d'exception).

Cette stratégie peut représenter presque toutes les relations possibles d'incompatibilité entre services. La seule contrainte, imposée par la nécessité de garantir un nombre fini d'états pour l'automate, est que tout service qui peut être interrompu par un autre service doit être incompatible avec une autre instance de lui-même. Si un service pertinent pour la résolution des conflits peut exister en un nombre non borné d'instances, il faut garder l'information à propos de ce nombre, ce qui rend non borné le nombre d'états de l'automate. Les services qui n'ont aucune incompatibilité, ou qui interrompent sans jamais être interrompus, peuvent être compatibles avec d'autres instances d'eux-mêmes, étant donné qu'il n'est pas nécessaire de savoir s'ils sont ou non en exécution pour résoudre les conflits.

À chaque transition entre états on associe les signaux que le contrôleur doit émettre vers l'extérieur (section 3.3.5). On illustre dans la figure 3.7 page suivante une transition de l'automate de notre exemple: comme le service `C` interrompt le service `E`, le groupe de services approprié passe de l'état `E` vers `IDLE` quand un service `C` démarre. Comme le service `C` n'utilise aucune ressource non partageable du robot (il n'est interrompu par aucun autre service) l'information à propos de son exécution n'est pas représentée dans l'automate.

La détermination du nombre minimal de groupes de services qui peut contenir l'information nécessaire à la résolution de tous les conflits potentiels peut être un peu moins évidente que dans l'exemple, principalement si le graphe d'incompatibilités contient des circuits ou des chemins en parallèle (chemins différents mais avec mêmes origines et destinations). Cependant, ce problème n'est pas fondamental, pour deux raisons:

1. Dans un robot réel, les relations d'incompatibilité entre services sont normalement

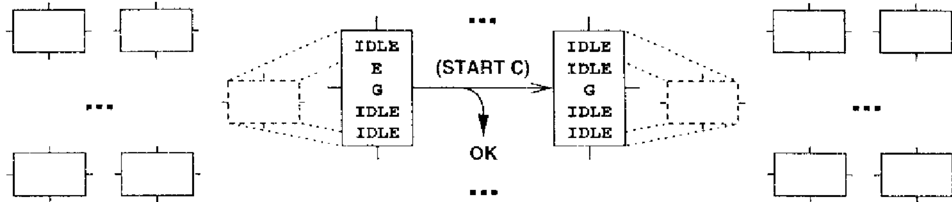


FIG. 3.7 *Une transition de l'automate qui représente le contrôleur de ressources*

simples. Comme les modules regroupent des fonctions qui ont des caractéristiques en commun (comme l'utilisation d'une même ressource), la situation la plus courante consiste à avoir un groupe de fonctions entièrement incompatibles entre elles (un sous-graphe complet et symétrique dans le graphe d'incompatibilités), avec d'autres fonctions qui n'ont aucune incompatibilité (normalement des fonctions de contrôle du module). Dans ce cas, il suffit d'avoir un groupe de services pour chaque module.

2. L'optimisation stricte en termes de nombre minimal de groupes de services n'est pas obligatoire, pourvu qu'on reste dans un nombre raisonnable d'informations à traiter à chaque cycle d'exécution de l'exécutif. La contrainte la plus importante à respecter (si les bornes temporelles ne sont pas dépassées) est que le contrôleur de ressources possède un comportement déterministe et correct pour chaque événement possible qu'on peut lui signaler, ce qu'on garantira avec l'utilisation du système Kheops (section 3.8).

3.6.2 La mise en œuvre du contrôleur de ressources

Le contrôle de l'utilisation des ressources est donc fait par changement d'état d'un automate qui représente les situations d'incompatibilité entre les services. Comme le grand nombre d'états de cet automate peut rendre le problème intraitable, on a décidé de ne pas les générer exhaustivement *a priori*. On a, à la place, adopté une représentation basée sur une fonction qui génère les transitions entre états. Si E_i est l'état de l'automate avant l'événement i et E_{i+1}, l'état suivant,

$$(E_{i+1}, \texttt{signal}) = f(E_i, \texttt{event}), \text{ où}$$

- `event` est l'événement qui a occasionné la transition de l'automate: demande de démarrage ou d'interruption d'une activité, signalement d'une fin d'exécution normale ou avec erreur, etc.; et
- f est la fonction de transition.
- `signal` représente le(s) signal(aux) que le contrôleur doit émettre.

Avec cette solution, on ne garde que l'état courant des modules: quand des événements arrivent, la fonction de transition est appelée pour calculer le nouvel état. Si on prend la

transition de la figure 3.7 page précédente comme exemple,

$$f(\begin{bmatrix} \mathtt{IDLE} \\ \mathtt{E} \\ \mathtt{G} \\ \mathtt{IDLE} \\ \mathtt{IDLE} \end{bmatrix}, \mathsf{STARTC}) = (\begin{bmatrix} \mathtt{IDLE} \\ \mathtt{IDLE} \\ \mathtt{G} \\ \mathtt{IDLE} \\ \mathtt{IDLE} \end{bmatrix}, \mathsf{OK})$$

La mise en œuvre de la fonction de transition f est une application où la quantité de connaissances à être incorporée est importante, ce qui suggère l'adoption d'un paradigme de programmation "basé sur la connaissance". Ce paradigme permet de spécifier les incompatibilités entre les services avec des règles proches de la connaissance que les concepteurs des modules peuvent fournir naturellement (**SI** je veux exécuter A, **ALORS** il faut interrompre B).

Il ne nous semble pas nécessaire de défendre les avantages de la spécification déclarative de la connaissance: on va simplement mentionner la modularité (due à un formalisme basé sur des règles), la rigueur (due à la logique associée) et la concision (à cause de l'absence d'informations de contrôle). Mais l'application des techniques d'intelligence artificielle (IA) dans un système réactif temps réel peut être contestée, car la performance n'est pas la grande qualité des systèmes d'IA.

Dans le cas des systèmes à base de règles de production, l'étape de filtrage ("*pattern matching*"), c'est à dire la détermination des règles applicables pour une situation donnée, peut occuper une grande partie du temps d'exécution du système. Même si quelques algorithmes ont été proposés pour améliorer la performance de cette étape, ils ne sont pas à eux seuls suffisants pour des applications en temps réel: il reste le problème du contrôle de l'inférence, étant donné que le nombre d'itérations[8] pour atteindre une solution est *a priori* non borné.

Cet ensemble de considérations, avec les contraintes associées, a mené à l'adoption du système Kheops pour la mise en œuvre du contrôleur de ressources.

3.7 Le système Kheops

Le système Kheops [Ghallab 88, Philippe 89, Gouyon 95] a été proposé comme une nouvelle approche pour la construction de systèmes basés sur la connaissance, destiné plutôt à des applications en temps réel. Dans Kheops un ensemble de règles propositionnelles monotones est transformé en un réseau décisionnel optimisé, de comportement déterministe. Une base de connaissances Kheops est formée par un ensemble fini de règles du type IF ...THEN ... qui font des inférences sur un ensemble d'attributs. À partir de certains attributs donnés comme entrées (*espace d'entrée*) le système Kheops déduit les valeurs d'autres attributs en sortie (*espace de sortie*). Dans un premier temps tous les enchaînements possibles des règles sont construits; la structure obtenue est ensuite optimisée et

8. Pour éviter une double utilisation du terme "cycle", on va le réserver aux cycles de déduction du système de production global, utilisant "itération" pour les cycles du moteur d'inférence. Un cycle peut occasionner plusieurs itérations avant d'aboutir à une réponse.

compactée. La finitude du réseau peut être assurée car les attributs ont logiquement des valeurs dans des ensembles finis.

Après une compilation, l'interprétation des règles est remplacée par une simple traversée d'un graphe. Un résultat fondamental pour des applications en temps réel est que cette procédure est caractérisée par une borne supérieure du temps d'exécution, déterminée par le chemin de la racine du graphe jusqu'à la feuille la plus profonde. Tous les attributs de l'espace observable sont testés au maximum une fois au long de n'importe quel chemin de la racine du graphe à un nœud feuille, ce qui revient à dire que la complexité de la procédure dépend du nombre de paramètres d'entrée, et non du nombre ou du contenu des règles.

L'enchaînement des règles doit mener à la détermination de tous les attributs de l'espace de sortie et aucun déclenchement de règle ne doit causer de contradictions. Cependant, pendant la compilation, les éventuelles configurations de l'espace d'entrée qui causent des inconsistances – un attribut réduit à un ensemble de valeurs possibles nul – ou des incomplétudes – tous les attributs de sortie ne sont pas déterminés – sont identifiées: il revient au programmeur de vérifier les problèmes causés par ces situations et de les réparer par ajout, suppression ou modification de règles.

Malgré le coût exponentiel de la procédure de compilation hors ligne, le système Kheops se présente comme une réponse aux deux principaux problèmes liés à l'efficacité d'un système de production déclaratif: l'accès aux conditions dans les règles est directe (il n'y a pas de filtrage) et le non déterminisme du système est apparent (le parcours dans le réseau est déterministe pour chaque entrée). Il peut donc être utilisé dans des applications en temps réel. La façon réactive dont le système fournit les sorties à partir des entrées à chaque cycle d'exécution permet de le considérer comme étant dans l'approche synchrone.

Un autre avantage majeur de Kheops pour des applications en temps réel est que, après génération du graphe de décision optimisé, le système produit un code source C équivalent au parcours de ce graphe. Ce code source peut être compilé et intégré dans l'application, où il ne restera aucune trace de la machine d'inférence Kheops pendant l'exécution.

La figure 3.8 page ci-contre reproduit un exemple classique [Ghallab 88] de programme Kheops et le graphe de décision correspondant. L'exemple montre les deux types d'actions autorisées dans la partie `THEN` des règles: des restrictions (`restrict`) sur le domaine de variation possible des variables ou des appels (`do`) à des fonctions C externes, qui peuvent établir ou non la valeur de certaines variables. Dans le graphe de décision, toute référence aux règles et à certaines variables intermédiaires (`diagn`) a disparu après la compilation.

L'annexe A explique en détails le formalisme qui sert de base à Kheops, la syntaxe et les algorithmes de compilation et d'optimisation du graphe de décision.

```
#define LOWDIFP 0
#define HIGHDIFP 10

inputs  pres1, pres2, deltaP, temp, drain,
        level, valve;

outputs proced;

r01: pres1<138 | pres1>=160 | deltaP >=4
 ==> restrict diagn oneof ["b1","b2","b3"];

r02: drain=="on"
 ==> restrict proced=="a3";

r03: diagn oneof ["b1","b2","b3"],
     drain =="off"
 ==> restrict diagn oneof ["b1","b2"];

r04: diagn oneof ["b1","b2"]
 ==> do {difP = pres2-pres1};

r05: diagn oneof ["b1","b2"], deltaP>=20
 ==> restrict diagn=="b2";

r06: diagn=="b1", valve=="open"
 ==> restrict proced=="a8";

r07: diagn=="b2", temp>=286
 ==> restrict proced=="a22";

r08: pres1>=138, pres1<160, deltaP<4,
     drain=="off"
 ==> restrict proced=="i3";

r09: diagn oneof ["b1","b2"], deltaP<20,
     difP>=HIGHDIFP
 ==> restrict proced=="a12";

r10: diagn oneof ["b1","b2"], pres2<40,
     difP<HIGHDIFP
 ==> restrict diagn=="b2";

r11: diagn oneof["b1","b2"], deltaP<20,
     pres2>=40, difP<HIGHDIFP
 ==> restrict diagn=="b1";

r12: diagn=="b2", temp<286, level=="high"
 ==> restrict proced=="a23";

r13: diagn=="b2", temp<286, level=="low"
 ==> restrict proced=="a21";

r14: diagn=="b1", difP<LOWDIFP,
     valve!="open"
 ==> restrict proced=="a11";

r15: diagn=="b1", difP>=LOWDIFP,
     valve!="open"
 ==> restrict proced=="a12";
```

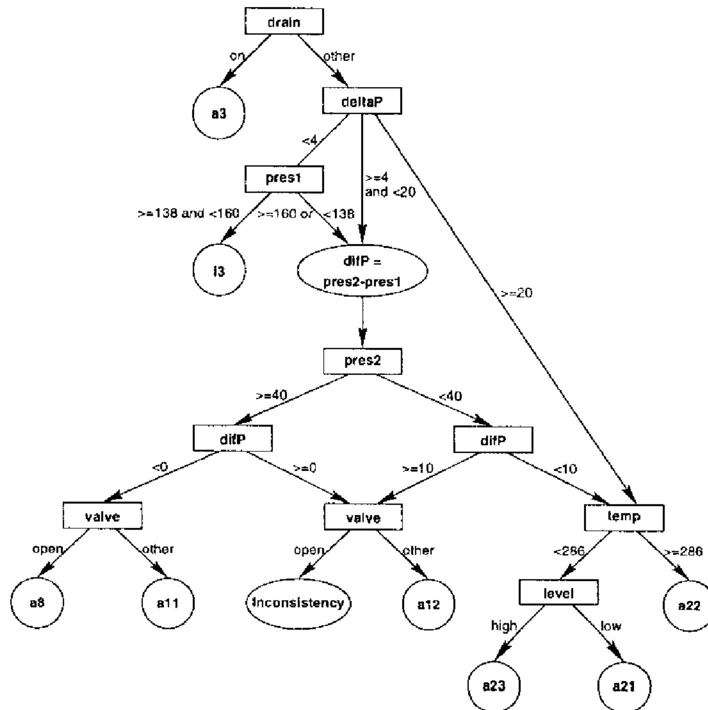


FIG. 3.8 – *Un exemple de base de règles* Kheops *et le graphe de décision correspondant*

3.8 Kheops et l'exécutif

Kheops a été utilisé pour mettre en œuvre la fonction de transition entre états (section 3.6.2 page 62) et, avec l'aide d'une partie extérieure écrite en C, les transitions du graphe de représentation des activités (fig. 3.3 page 52). Pour pouvoir mieux présenter certains détails, nous allons les illustrer au moyen d'une base de connaissances développée pour un système générique représenté dans la figure 3.9. Ce système, composé de seulement deux modules, a été défini de manière à contenir plusieurs types de conflits entre services.

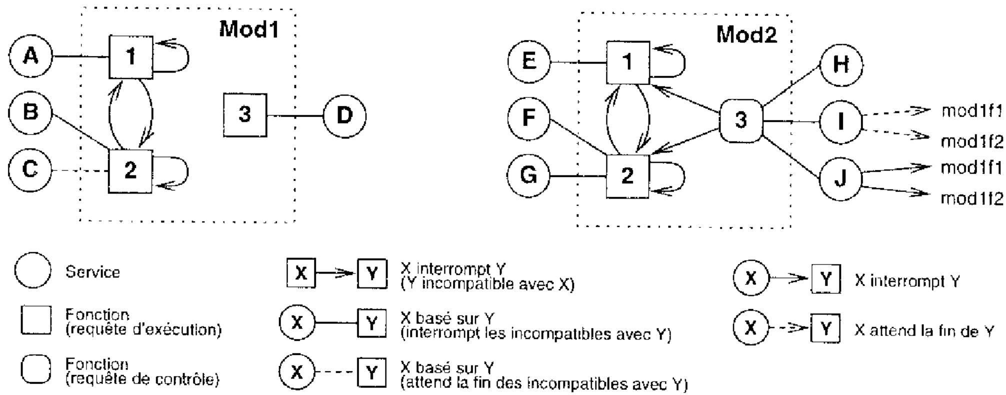


FIG. 3.9 – *Niveau fonctionnel d'un système générique*

Le premier module a trois fonctions: `mod1f1`, `mod1f2` et `mod1f3`. Une seule instance des activités `mod1f1` et `mod1f2` peut exister à un moment donné, alors que `mod1f3` peut être exécutée avec plusieurs instances et simultanément avec `mod1f1` et `mod1f2`. Le module `mod2` a aussi trois fonctions, avec `mod2f1` et `mod2f2` totalement incompatibles entre elles. `mod2f3` est une fonction activée par une requête de contrôle (de durée considérée nulle et donc ininterruptible), mais qui interrompt toute instance d'une autre activité dans son module. En plus, selon le contexte, l'exécution de `mod2f3` peut ou non exiger l'interruption de `mod1f1` et `mod1f2`.

On veut offrir 10 services (`A` à `J`) au niveau décisionnel, avec les caractéristiques présentées dans le tableau 3.1 page ci-contre. On peut remarquer que la fonction `mod2f3` est utilisée par trois services différents (`H`, `I` et `J`), chaque fois avec un comportement différent face aux fonctions du premier module. Les services `F` et `G` sont identiques du point de vue des conflits, et diffèrent seulement en termes des paramètres de la requête.

À partir des incompatibilités entre les fonctions et des fonctions utilisées par chaque service, on peut établir le graphe des incompatibilités entre les services, représenté dans la figure 3.10 page suivante. On indique en lignes pointillées les incompatibilités où le service attend la terminaison du service en conflit, au lieu de l'interrompre, mais cette distinction n'a aucune importance pour la détermination de la quantité de groupes de service nécessaires et des services intégrants chaque groupe.

Par regroupement des sous-graphes complets et symétriques, il dévient évident que

TAB. 3.1 – *Les services du système générique*

Service	fonction	Incompatible avec	Action
A	mod1f1	mod1f1 mod1f2	Interrompre
B	mod1f2	mod1f1 mod1f2	Interrompre
C	mod1f2	mod1f1 mod1f2	Attendre la fin
D	mod1f3		
E	mod2f1	mod2f1 mod2f2	Interrompre
F	mod2f2	mod2f1 mod2f2	Interrompre
G	mod2f2	mod2f1 mod2f2	Interrompre
H	mod2f3	mod2f1 mod2f2	Interrompre
I	mod2f3	mod1f1 mod1f2	Attendre la fin
		mod2f1 mod2f2	Interrompre
J	mod2f3	mod1f1 mod1f2 mod2f1 mod2f2	Interrompre

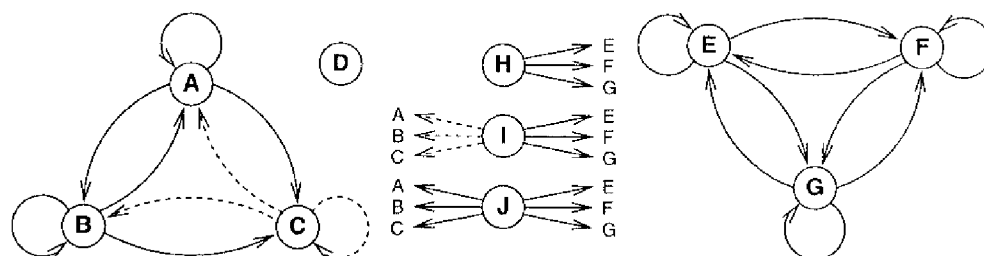


FIG. 3.10 – *Graphe d'incompatibilités pour le système générique*

deux groupes de services, ABC et EFG, sont nécessaires pour la résolution des conflits.

3.8.1 Le contrôleur de ressources

La base de connaissances doit faire la transition entre les états de l'automate, selon les signaux qu'elle reçoit, et émettre les réponses appropriées. Les variables d'entrée[9], qui indiquent le signal qui a occasionné l'appel au contrôleur de ressources, sont:

- `service` – sont reconnus tous les services de la liste de description des services.
- `event` – le type d'événement:
  - `START` – sollicitation de démarrage.
  - `INTERM` – arrivée de la réplique intermédiaire.
  - `END` – fin d'exécution. Pour le contrôleur de ressources, les fins d'exécution normales, par interruption ou avec erreur sont équivalentes: seule la libération de la ressource compte.

9. La valeur de ces variables est fournie soit par le gestionnaire de requêtes, soit par le gestionnaire de répliques, au moment où ils sollicitent un cycle d'exécution du contrôleur de ressources.

- Comme le fonctionnement est cyclique, la valeur dans le cycle précédant des variables de sortie *group*`state` (qui représentent l'état de l'automate) sont des entrées implicites pour le cycle courant.

Les variables de sortie, qui représentent l'état de l'automate et le signal émis par le contrôleur destiné au composant qui a sollicité le cycle d'exécution, sont:

- `status` – la réponse du contrôleur à l'événement:
  - `OK` - autorisation accordée ou signalement pris en compte.
  - `WAIT` - autorisation non accordée pour le moment.
  - `REFUSE` - autorisation refusée.
  - `ERROR` - événement non reconnu ou impossible dans le contexte actuel.
- *group*`state` – le service du groupe *group* qui est en exécution ou `IDLE`, si aucun.

Le contrôleur, en plus d'arbitrer et suivre l'utilisation des ressources, vérifie la cohérence du système: tout événement non attendu ou incorrecte est signalé (avec `status` = `ERROR`), ce qui devra mettre l'exécutif dans un état d'erreur général où toute nouvelle requête est refusée, en attente d'une intervention de l'opérateur.

Les règles

On présente ici quelques unes des règles le plus importantes de la base de connaissance qui décrit le contrôleur de ressources du système générique.

Si un groupe est `IDLE`, il le reste jusqu'à ce que l'arrivée d'une réplique intermédiaire d'un de ses services le change d'état.

```
intermB:
  event == "INTERM",
  service == "B",
  @ABCstate noneof ["A","B","C"]
==>
  restrict status == "OK",
  restrict ABCstate == "B";
```

Un service qui attend la fin d'autres services peut démarrer si le groupe est `IDLE`; ...

```
CstartWhenIdle:
  event == "START",
  service == "C",
  @ABCstate noneof ["A","B","C"]
==>
  restrict status == "OK";
```

sinon, il doit attendre.

```
CstartWhenNotIdle:
  event == "START",
  service == "C",
  @ABCstate oneof ["A","B","C"]
==>
  restrict status == "WAIT";
```

Pour les services qui ne font pas partie d'un groupe, l'arrivée des répliques (intermédiaire et finale) n'occasionne aucun changement.

```
replyD:
  event oneof ["INTERM","END"],
  service == "D"
==>
  restrict status == "OK";
```

La requête du service D, qui n'est incompatible avec aucun autre, est toujours acceptée.

```
Dstart:
  event == "START",
  service == "D"
==>
  restrict status == "OK";
```

Les services qui n'attendent la fin d'aucun autre sont normalement acceptés. S'il n'y a pas de services du même groupe en exécution, il n'y a rien à faire; ...

```
EFGHstartIdle:
  event == "START",
  service oneof ["E","F","G","H"],
  @EFGstate noneof ["E","F","G"]
==>
  restrict status == "OK";
```

sinon, le contrôleur change (par appel à une fonction extérieure) l'état de l'activité associée au service déjà en exécution en INTER.

```
EFGHstartNotIdle:
  event == "START",
  service oneof ["E","F","G","H"},
  @EFGstate oneof ["E","F","G"]
==>
  do {changeInterState(@EFGstate)},
  restrict status == "OK";
```

Le service I attend la fin des services du groupe ABC, ...

```
IstartWhenABCnotIdle:
  event == "START",
  service == "I",
  @ABCstate oneof ["A","B","C"]
==>
  restrict status == "WAIT";
```

mais pas de ceux du groupe EFG. Comme ce sont des services du même module que I, il n'est pas nécessaire de les interrompre explicitement, mais seulement de changer l'état de l'activité.

```
IstartWhenABCidleEFGnotIdle:
  event == "START",
  service == "I",
  @ABCstate noneof ["A","B","C"],
  @EFGstate oneof ["E","F","G"]
==>
  do {changeInterState(@EFGstate)},
  restrict status == "OK";
```

Pour démarrer J si ABC n'est pas IDLE, le contrôleur envoie une requête explicite d'interruption de l'activité. Même si le service J n'attend aucun service, le contrôleur doit quand même répondre WAIT, parce que l'interruption n'est pas instantanée.

```
JstartWhenABCnotIdle:
  event == "START",
  service == "J",
  @ABCstate oneof ["A","B","C"]
==>
  do {changeInterState(@ABCstate)},
  do {sendAbortRequest(@ABCstate)},
  restrict status == "WAIT";
```

Les groupes reviennent à IDLE après la fin de l'exécution de leur services.

```
endABC:
  event == "END",
  (  (service == "A", @ABCstate == "A")  |
     (service == "B", @ABCstate == "B")  |
     (service == "C", @ABCstate == "C")  )
==>
  restrict status == "OK",
  restrict ABCstate noneof ["A","B","C"];
```

Plusieurs événements incorrects sont signalés. Cette vérification de consistance est correcte, mais n'est pas complète pour les services qui ne font pas partie d'un groupe.

Les événements qui ne peuvent pas avoir lieu pour un service qui n'est pas en exécution.

```
eventABCnotRunning:
  event oneof ["END","ABORT","ZOMBIE"],
  (  (service == "A", @ABCstate != "A")   |
     (service == "B", @ABCstate != "B")   |
     (service == "C", @ABCstate != "C")   )
==>
  restrict status == "ERROR";
```

Des répliques intermédiaires à des requêtes de contrôle.

```
intermControl:
  event == "INTERM",
  service oneof ["H","I","J"]
==>
  restrict status == "ERROR";
```

Des répliques intermédiaires pour un service qui fait partie d'un groupe de services qui n'est pas IDLE.

```
intermEFGnotIdle:
  event == "INTERM",
  service oneof ["E","F","G"],
  @EFGstate oneof ["E","F","G"]
==>
  restrict status == "ERROR";
```

Pour comprendre la raison de l'impossibilité de cette dernière situation, il suffit de se rappeler que le contrôleur du module, lors de la réception d'une requête, interrompt d'abord les activités en conflit avant de retourner la réplique intermédiaire. Quand l'interruption sera effectuée, le contrôleur envoie, presque simultanément mais toujours dans cet ordre, d'abord la réplique finale de l'activité interrompue et ensuite la réplique intermédiaire de l'interruptrice. Le contrôleur va donc traiter en premier la réplique finale, qui rend IDLE le groupe de services, et juste après la réplique intermédiaire.

3.8.2 Les gestionnaires de requêtes et de répliques

Les règles Kheops qui décrivent l'évolution du graphe de représentation des activités sont génériques et indépendantes des services offerts par l'exécutif. Les règles du contrôleur de ressources et des gestionnaires de requêtes et de répliques ont été compilées ensemble et forment un grand graphe de décision, qui exécute la plupart des activités de chaque cycle de l'exécutif. Des parties codées en C se chargent de la communication finale avec le niveau tâche et les modules. Un cycle typique de l'exécutif déroule l'algorithme suivant:

1. Attendre un événement (requête ou réplique ou réveil périodique quand il y a des activités en attente).
2. Pour chaque activité dans la liste d'activités:
   (a) Lire une éventuelle réplique envoyée par le module.
   (b) Parcourir le graphe de décision généré par Kheops,
   (c) Exécuter les actions préconisées.

La base de règles Kheops générale (qui code le fonctionnement du contrôleur de ressources et des gestionnaires) prend comme entrées les paramètres suivants:

`service` - le service concerné (INTERRUPT pour les requêtes d'interruption).

`state` – l'état actuel de l'activité.

`stateInt` – l'état actuel de l'activité à interrompre. Ce paramètre n'est pris en compte que dans le cas des requêtes d'interruption.
`WAIT INIT INTER EXEC ZOMBIE ETHER`

`reply` – la situation par rapport aux répliques reçues du module pour cette activité:
`WAITING-INTERMED-REPLY INTERMED-REPLY-TIMEOUT WAITING-FINAL-REPLY FINAL-REPLY-TIMEOUT FINAL-REPLY-OK ERROR`

`bilanIn` – le (type de) bilan reçu avec la réplique finale. N'est pris en compte que si la réplique finale a été reçue.
`ZOMBIE INTERRUPTED OK` autre

`timeout` – si le temps d'attente maximal a été dépassé. N'est pas pris en compte pour les requêtes d'interruption (qui ne sont jamais mises en attente).

`zombieIn` – s'il y a une activité en `ZOMBIE` dans le module de la requête. N'est pas considéré pour les requêtes d'interruption (qui ne sont pas refusées même dans ce cas).

Les variables de sortie produites sont:

`newState` – nouvel état de l'activité.

`newStateInt` – nouvel état de l'activité à interrompre.

`sendRqst` - s'il faut envoyer la requête associée à ce service aux modules.

`sendTask` – s'il faut envoyer un message au niveau tâche. À partir du type d'activité (interruption ou non), l'exécutif déduit le type (avertissement ou réplique) du message.

`bilanOut` – le bilan à retourner au niveau tâche (si un message doit être envoyé). Si `OK`, indique qu'il faut retourner le bilan original envoyé par le module. Sinon, donne la valeur du bilan.
`OK BAD-ID TIMEOUT REFUSED`

`zombieOut` – pour une requête d'interruption, indique s'il faut sortir le module de l'état `ZOMBIE`; pour une requête d'exécution, s'il faut le mettre en état `ZOMBIE`.

`fatalError` – signale une erreur fatale (incohérence). L'exécutif se met en état `ERROR`.

Pour calculer les variables de sortie, l'ensemble de règles Kheops calcule certaines variables intermédiaires (comme `status`, la sortie du contrôleur de ressources). Dans la figure 3.11 page suivante on présente des exemples de règles Kheops des gestionnaires de requêtes et de répliques qui font évoluer l'état des activités.

3.9 La communication avec le niveau tâche

L'exécutif communique avec le niveau décisionnel de deux manières différentes:

- Les requêtes du niveau tâche sont envoyées par messages. Chaque message contient le nom du service désiré et un identificateur unique qui sera utilisé dans les communications futures à propos de ce service. Les services ne prennent pas de paramètres, excepté la requête d'interruption d'un service (`INT`), où l'identificateur correspond au service à interrompre.

```
initFatalError:
  state == "INIT",
  reply oneof ["INTERM-TIMEOUT","WAITING-INTERM"],
  !interm
==>
  restrict fatalError;
```

```
WaitToInit:
  state == "WAIT",
  status == "OK",
  !zombieIn
==>
  restrict newState == "INIT",
  restrict sendRqst;
```

```
normalEnd:
  state == "EXEC",
  reply == "FINAL-REPLY-OK",
  bilanIn noneof ["INTERRUPTED","ZOMBIE"],
  status == "OK"
==>
  restrict bilanOut == "OK",
  restrict sendTask;
```

```
zombieInterrupted:
  state == "INIT",
  reply = "FINAL-REPLY-OK",
  bilanIn == "OK",
  intState == "ZOMBIE"
==>
  newState == "ETHER",
  newIntState == "ETHER",
  zombieOut;
```

FIG. 3.11 – *Quelques règles des gestionnaires de requêtes et de répliques*

- Les répliques sont aussi envoyées par messages, mais ils agissent en altérant la base de données du niveau tâche. Après envoi d'une requête, la procédure impliquée surveille l'évolution de la base de données en attente de faits qui concernent le service demandé. Il n'y a pas d'ambiguïtés car tous les faits contiennent l'identificateur du service auquel ils sont liés. Il y a deux types principaux de faits:

  (REPLY *id result*) – fin d'exécution (réplique finale) du service *id*. *result* indique si l'exécution a été normale ou avec erreur. Dans le premier cas, *result* contient OK ou, si la fonction la fournit, une description logique du résultat retourné; en cas d'erreur, *result* donne la catégorie de l'erreur (TIMEOUT, INT, etc.).

  (WARNING *serv id type*) – signale un événement *type* concernant le service *id* (*serv*). Si *type* == ZOMBIE, le service est figé, exigeant une interruption explicite avant d'envoyer sa réplique finale et libérer le module.

3.10 Conclusion

Le niveau décisionnel peut utiliser la bibliothèque de services offerte par l'exécutif pour accomplir la mission du robot. On ne va pas présenter dans ce mémoire une mise en œuvre particulière du niveau tâche, étant donné qu'elle est dépendante de la mission, mais dans le chapitre suivant on montre les procédures de base qui permettent au système PRS (sur lequel se base le niveau tâche) d'utiliser les services de l'exécutif.

Une conséquence importante du fait que les données "brutes" restent locales au niveau fonctionnel est qu'il devient envisageable de concevoir un niveau tâche qui n'utilise que des variables avec un domaine de variation fermé. Dans ce cas, PRS peut être réduit à un sous ensemble pour lequel une vérification logique (et même temporelle) est possible. Le chapitre suivant présente des résultats relatifs à la vérification de ce sous-ensemble de PRS avec les réseaux de Pétri.

Chapitre 4

Le niveau tâche

Les modules et l'exécutif sont les composants génériques de l'architecture de contrôle. Idéalement, ils doivent fournir un ensemble de services indépendants de la tâche à accomplir, que le niveau décisionnel combinera de façon à effectuer la mission que l'on attend du robot. Le niveau tâche est donc le premier niveau (et le seul dans plusieurs missions) où le concepteur définira les aspects spécifiques à la tâche globale.

Notre travail n'a pas la prétention de fournir un modèle global pour un niveau tâche dans le cadre de l'architecture LAAS, mais simplement d'indiquer la manière d'intégrer les services offerts par le niveau fonctionnel dans un ensemble de procédures PRS. Pour cela, on commence par donner un petit aperçu du système PRS (section 4.1). Ensuite, on présente quelques procédures de base essentielles pour la mise en œuvre du niveau tâche (section 4.2).

Comme la politique de distribution de données que nous avons proposé réserve en priorité la manipulation des données non dénombrables au niveau fonctionnel, la totalité ou une bonne partie des données manipulées par le niveau tâche est dénombrable, voire fermée (prenant ses valeurs dans un univers de possibilités fini et connu d'avance). Pour le sous-ensemble de PRS qui ne manipule que des variables fermées, on peut trouver une équivalence avec un réseau de Pétri coloré, ce qui permet une vérification logique d'une partie du niveau tâche.

Dans la section 4.3 on récapitule rapidement les concepts de base des réseaux de Pétri colorés. On définit ensuite (section 4.4) le sous-ensemble du langage PRS qui admet une équivalence avec les réseaux et les règles de conversion. Finalement, dans la section 4.5, on montre des exemples des propriétés logiques qu'on peut prouver pour la partie du niveau tâche pour laquelle l'équivalence existe.

4.1 Le système PRS

PRS est composé d'un ensemble d'outils et de méthodes pour représenter et exécuter plans et procédures. PRS est bien adapté aux contrôle de systèmes complexes en temps réel [Ingrand 92], en particulier pour les robots mobiles autonomes: dans plusieurs travaux réalisés au LAAS [Ingrand 96, Ingrand 95], il est utilisé pour la supervision de robots avec un large degré d'autonomie. Une étude complète de la représentation procédurale de la

connaissance dépasse l'objectif de ce travail: on citera seulement les travaux de référence de Georgeff et Lansky [Georgeff 86] et d'Ingrand *et al* [Ingrand 92], où peut être trouvée une information plus complète sur ces méthodes.

Certaines caractéristiques de PRS seront mieux expliquées dans la section 4.4, lors de la définition de leur représentation par des réseaux de Pétri. Pour le moment, et d'une façon résumée, on peut dire qu'un agent PRS consiste en:

Une base de données: contient des faits qui représentent le monde vu par le système; elle est constamment et automatiquement mise à jour quand des nouveaux événements (provenant de l'environnement, de PRS lui-même ou de l'utilisateur) arrivent.

L'ensemble des buts courants: en PRS, les buts décrivent des objectifs et renseignent sur la façon de les atteindre. Le but pour obtenir une certaine condition C s'écrit `(! C)`; `(? C)`, pour tester une condition; `(^ C)`, pour attendre jusqu'à qu'une condition soit vraie; `(# C)`, pour préserver passivement C; et `(% C)`, pour maintenir activement C[1]. Deux autres opérateurs permettent d'établir une information dans la base de données `(=> C)` et de l'enlever de la base de données `(~> C)`.

Bibliothèque de procédures: chaque procédure décrit une séquence particulière d'actions et de tests qui peuvent être exécutés pour accomplir des buts donnés ou pour réagir à certaines situations.

Graphe d'intentions: un ensemble dynamique de procédures en exécution, structuré sous la forme d'un graphe, où le système maintient la trace de l'état d'exécution des procédures essayées et de leurs sous-buts postés.

Un interpréteur (machine d'inférence) manipule ces composants. Il reçoit les nouveaux événements, sélectionne les procédures appropriées pour la situation courante, les place éventuellement dans le graphe d'intentions et les exécute.

La connaissance qui définit comment accomplir des buts ou réagir à des situations est représentée par des procédures déclaratives appelées KAs (*Knowledge Areas*), comme celle de la figure 4.1 page suivante. Chaque KA a un corps, qui décrit les pas de la procédure, et des conditions d'invocation qui spécifient sur quelles conditions la KA est utilisable.

Le corps de la KA est un réseau graphique qui peut être vu comme un plan. C-PRS permet aussi d'écrire des KAs avec des instructions en mode texte, mais qui se ramènent à des KAs graphiques. Chaque arc du réseau est étiqueté avec un but à atteindre par le système. Les conditions d'invocation ont deux parties, qui doivent toutes les deux être satisfaites pour que la KA puisse être exécutée:

Conditions de déclenchement (champ `INVOCATION` – événements (exprimés par une expression logique) qui doivent se produire pour que la KA soit considérée pour l'exécution. Les événements sont normalement l'apparition d'un nouveau but (comme dans l'exemple de la fig. 4.1 page ci-contre), un changement dans la base de données (pour les KAs déclenchées par des faits) ou des combinaisons des deux.

Contexte (champ `CONTEXT`) – expression logique qui doit être vraie pour que, si les événements de déclenchement ont lieu, le corps de la KA puisse être exécuté.

1. Il y a une autre notation, définie en C-PRS , qui permet des mots plus explicites à la place des symboles. `(? C)`, par exemple, devient `(test C)`.

Long Range Displacement

```
INVOCATION:
(! (GOTO-FAR-POSITION))

CONTEXT:
(? (& (IS-LONG-RANGE-DISPLACEMENT)
      (NUMBER-OF-DISPLACEMENTS $I)
      (< I$ 5)))

EFFECTS:
((~> (NUMBER-OF-DISPLACEMENTS $I))
 (=> (NUMBER-OF-DISPLACEMENTS (+ $I 1))))

DOCUMENTATION:
"Cette KA execute des longs deplacements,
 en choisissant des sous-buts
 intermediaires jusqu'a que le robot
 soit suffisament proche pour faire un
 deplacement court"
```

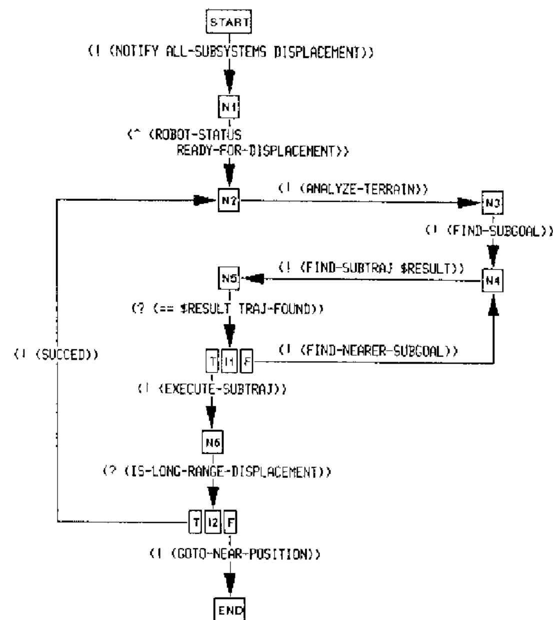


FIG. 4.1 – *Un exemple de procédure (KA)* PRS

Le corps de la KA décrit ce qu'il faut faire si la KA est choisie pour exécution. L'exécution commence au nœud **START** et se poursuit en suivant les arcs à travers le réseau. L'exécution se termine si elle atteint un nœud terminal (un nœud sans arcs partants). Si plus d'un arc part d'un nœud, n'importe lequel d'entre eux peut être traversé[2]. Pour traverser un arc, le système doit soit contrôler que le but a déjà été atteint (par inspection de la base de données), soit dérouler une KA qui atteint le but associé à l'arc. Si le système ne réussit à atteindre aucun des buts associés aux arcs qui partent d'un nœud, la KA entière échoue.

Les conditions peuvent contenir des variables à unifier. Par exemple, on peut utiliser le prédicat **BATTERY-STATE** pour tester directement si le niveau de la batterie est bas, avec **(? (BATTERY-STATE LOW))**, ou pour attribuer à la variable **$STATE** l'état actuel de la batterie, avec **(? (BATTERY-STATE $STATE))**. Dans les deux cas on suppose que la base de données contient l'information en question. Il y a deux types de variables en PRS : les variables logiques $\$var$ ont le comportement classique des variables en programmation logique (une fois unifiées, elles ne changent plus de valeur), alors que la valeur des variables de programme **@***var* peut être modifiée à tout instant.

La fin d'exécution réussie d'une KA déclenchée par un but a comme effet implicite la satisfaction de ce but. L'utilisateur peut ajouter d'autres conséquences à l'exécution de la KA avec le champ optionnel **EFFECTS** (fig. 4.1): il s'agit d'une liste de faits qui seront conclus ou éliminés dans la base de faits en cas d'exécution réussie.

2. Sauf, comme on verra par la suite, pour les nœud qui lancent des branches en parallèle.

Certaines KAs n'ont pas de corps (comme celle de la figure 4.2). Elles correspondent aux KAs primitives et sont associées à des actions de base que le système sait exécuter directement. Dans l'exemple de la figure 4.1 page précédente, le postage des buts `FIND SUBTRAJ` doit faire appel à des fonctions externes qui savent exécuter les actions désirées. L'utilisateur peut aussi définir ses propres prédicats évaluables, qui sont testés par exécution d'une fonction externe et non par consultation de la base de données (`IS-LONGE-RANGE-DISPLACEMENT` dans l'exemple de la figure 4.1 page précédente, en occurrence un prédicat sans argument). L'exécution de toutes les KAs doit se ramener à l'exécution d'une séquence de primitives que PRS sait exécuter directement (KAs primitives, prédicats évaluables, opérations dans la base de données, etc.).

Send Request

INVOCATION:
```
(! (SEND-REQUEST $REQUEST $ID))
```

ACTION:
```
(*=* $ID
    (SEND-REQUEST $REQUEST))
```

DOCUMENTATION:
```
"Envoie la requete $REQUEST a l'executif.
L'action retourne l'ID de la requete."
```

FIG. 4.2 – *Un exemple de KA primitive*

PRS prévoit aussi l'existence de meta-KAs, qui manipulent les assertions, les buts et les intentions de PRS lui-même. Des meta-KAs typiques sont, par exemple, celles qui contiennent des méthodes pour choisir entre des multiples KAs applicables ou pour restreindre le raisonnement afin de respecter des contraintes temporelles.

4.2 La mise en œuvre du niveau tâche

4.2.1 La communication avec l'exécutif

La communication entre PRS et l'exécutif est basée principalement sur les mécanismes d'envoi et de réception de messages existants en PRS. Dans la figure 4.3 page suivante on présente une version simplifiée de la procédure générique d'exécution d'une requête de service.

L'exécution commence par l'envoi de la requête. Le but `(! (SEND-REQUEST ...))` est satisfait par une KA primitive (fig. 4.2). Cette KA fait appel à une fonction externe qui attribue un identificateur unique à la requête et les envoie (la requête et l'identificateur) à l'exécutif.

La réaction de l'exécutif se traduira par l'envoi d'un autre message, qui sera enregistré dans la base de données de PRS comme un fait. La procédure se met donc en attente de l'apparition d'un de ces deux faits dans la base de données:

`(REPLY $ID $RESULT)` – La réplique finale du service. `$RESULT` retourne le bilan d'exécution.

`(WARN $REQUEST $ID ZOMBIE)` – Le service s'est terminé en état `ZOMBIE`. Dans ce cas, après émission d'un avertissement à l'opérateur, la procédure envoie une demande d'interruption du service et se remet en attente de sa réplique finale.

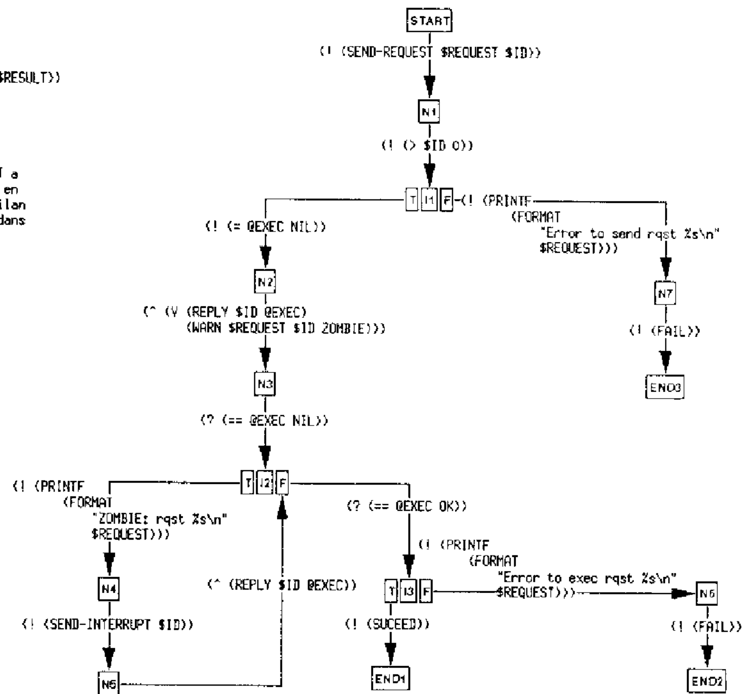


FIG. 4.3 – *Exemple de procédure de communication entre l'exécutif et* PRS

Les procédures du niveau tâche peuvent accéder aux services du niveau fonctionnel soit en postant directement des buts `(! (EXEC-REQUEST` ***requête***`))`, soit en utilisant d'autres KAs qui le font. Ces KAs spécifiques peuvent contenir un premier traitement du bilan d'exécution, adapté à chaque service, comme par exemple (fig. 4.4 page suivante) une tentative de reprise en cas d'insuffisance algorithmique[3].

4.2.2 La manipulation des données

D'une façon générale, dans le modèle d'architecture qu'on propose, le niveau tâche manipule presque exclusivement des données dénombrables (des entiers, des variables symboliques, etc.). Cela est possible parce que:

- toutes les variables complexes sont déclarées soit comme internes à un module ou exportables dans un poster, soit comme des variables au niveau de l'exécutif; et
- l'existence du dictionnaire de services au niveau de l'exécutif permet l'utilisation des fonctions des modules avec différents paramètres, sans que le niveau décisionnel ait besoin de les manipuler.

3. Si un service n'a pas réussi à trouver une solution, on essaye un autre service qui traite le même problème avec une méthode différente.

Find Trajectory to Subgoal

INVOCATION:
(! (FIND-SUBTRAJ $RESULT))

DOCUMENTATION:
"La procedure essaye de trouver une
trajectoire pour le robot en mode
de locomotion normal. Si cela echoue,
elle essaye de trouver un chemin
realisable en mode de locomotion
peristaltique"

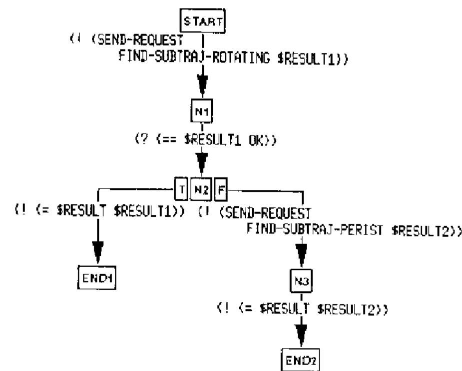


FIG. 4.4 – *Une KA de traitement spécifique d'un service*

Dans certaines situations, cependant, le niveau tâche peut avoir besoin de manipuler ponctuellement des informations numériques. Cela est possible grâce à la définition des requêtes de service spécifiques `GET-PARAM` et `SET-PARAM`, qui sont utilisées dans des expression du type `(GET-PARAM ROBOT-POSITION $VALUE)` et `(SET-PARAM MAX-SPEED $VALUE)`. L'exécutif répond à ces deux requêtes de manière particulière: soit il prend en compte le paramètre envoyé, soit il retourne la valeur désirée.

Si les manipulations de variables non dénombrables sont faites avec modération, une large partie du niveau tâche ne contient que des valeurs prédéfinies dans un univers connu. Pour ce sous-ensemble, on peut définir un modèle équivalent, basé sur les réseaux de Pétri colorés, à partir duquel certaines propriétés de comportement (comme la vivacité et la répétabilité) peuvent être prouvées.

4.3 Les réseaux de Pétri colorés

On ne va pas présenter en détails la théorie des réseaux de Pétri colorés (RPCs), mais seulement faire quelques remarques intuitives à partir de l'exemple de la figure 4.5 page ci-contre [Jensen 90], qui représente un cas simplifié d'allocation de ressources. Des formalisations plus précises à propos des réseaux de Pétri en général et des RPCs en particulier peuvent être trouvées dans la littérature [Murata 89, Jensen 94].

Dans les RPCs, chaque place peut contenir plusieurs jetons, chacun d'entre eux ayant une valeur associée qu'on appelle la couleur du jeton. Dans l'exemple, la couleur des trois jetons initialement présents dans la place `A` est un doublet `(q,0)`. Chaque place a un type associé, qui fixe les couleurs de jetons qu'elle peut contenir. La place `S`, par exemple, est du type `E`, ce qui implique qu'elle ne peut contenir que des jetons `(e)` (des jetons sans couleur). Les variables ont aussi des types associés, qui définissent leurs domaines.

Les étiquettes des arcs indiquent combien de jetons de chaque couleur les transitions consomment et libèrent. L'étiquette associée à une transition, si elle existe, représente une condition supplémentaire (en plus de l'existence des jetons appropriés dans les places d'entrée) à être satisfaite pour pouvoir tirer la transition. Pour que la transition soit validée,

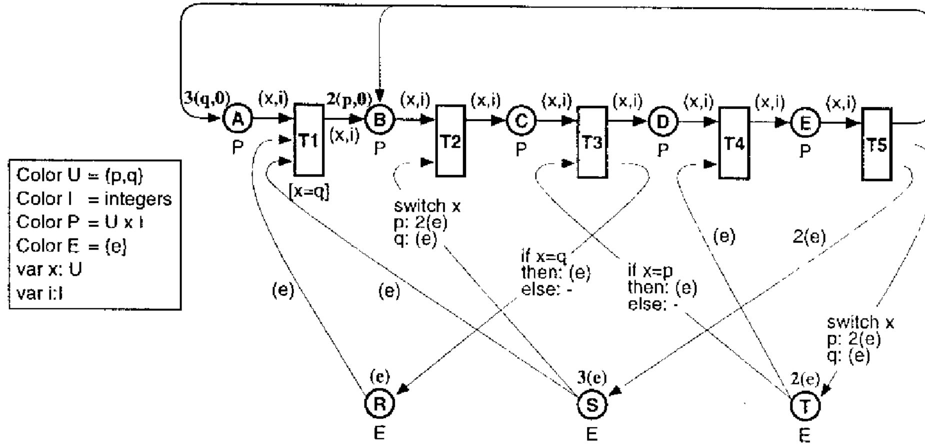


FIG. 4.5 – *Un exemple de réseau de Pétri coloré*

quand la même variable apparaît plus d'une fois dans les étiquettes de la transition et/ou des arcs associés, toutes les occurrences doivent pouvoir être unifiées à la même couleur. Pour un marquage donné, une transition peut être validée avec plusieurs unifications différentes, mais seulement une des possibilités se concrétisera à chaque tir. Dans la figure 4.6 on donne un exemple de tir d'une transition.

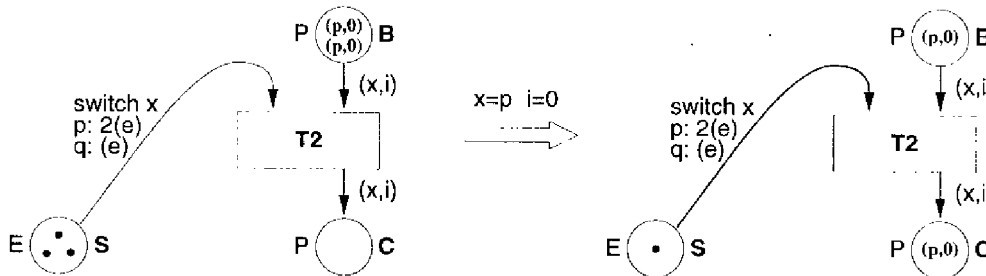


FIG. 4.6 – *Tir de transition dans un réseau de Pétri coloré*

Le domaine d'existence d'une variable dans un RPC est la transition en contacte avec l'arc où la variable est déclarée. Dans le RPC de la figure 4.5, par exemple, il n'y a aucun lien entre les variables `X` et `i` de l'arc qui lie `T3` à `D` et les variables `X` et `i` de l'arc qui lie `D` à `T4`, parce que ces variables sont déclarées dans des arcs qui n'ont pas de transition en commun.

Un RPC peut être considéré comme une version structurée d'un réseau de Pétri régulier si le nombre de couleurs est fini. Dans ce cas, le RPC peut être converti en un réseau classique en générant pour chaque place autant de places que de couleurs possibles pour ses jetons, et pour chaque transition autant de transitions que de manières différentes de la tirer. Comme conséquence naturelle, les propriétés qu'on peut prouver avec un réseau de Pétri classique peuvent aussi être prouvées avec un RPC à nombre de couleurs finis.

4.4 PRS et les réseaux de Pétri

Initialement, on peut se poser la question de l'intérêt d'utiliser spécifiquement les réseaux de Pétri comme outil de vérification d'un sous-ensemble de PRS. La réponse est liée à la versatilité autorisée dans PRS. Il faut un modèle capable de représenter le parallélisme et le non déterminisme et de combiner des parties procédurales avec une certaine logique déclarative.

Même si les réseaux de Pétri ne sont pas capables de modéliser toutes les possibilités de PRS, notre opinion est que, parmi toutes les méthodes habituellement utilisées en vérification et validation [Lee 94], les RP sont le modèle qui peut couvrir le sous-ensemble le plus complet des fonctionnalités de PRS, principalement dans le cadre de notre architecture de contrôle pour robots. Les réseaux de Pétri sont intrinsèquement capables de modéliser les choix non déterministes et les aspects procéduraux, comme le parallélisme, et peuvent être utilisés comme outils de vérification de systèmes déclaratifs [Liu 91], ce qui les rend adaptés à la double nature (parties procédurales et déclaratives) des procédures PRS.

La finalité majeure de trouver une représentation par réseaux de Pétri est de pouvoir prouver un certain nombre de propriétés pour les procédures PRS, comme la vivacité (absence de blocages). Il est donc fondamental d'utiliser un type de réseau de Pétri pour lequel la possibilité d'analyse formelle existe. Les RPCs sont peut-être le modèle avec la meilleure capacité d'expression entre ceux qui admettent une équivalence avec les réseaux de Pétri classiques (réseaux place/transition). D'autres extensions, comme les arcs inhibiteurs, augmentent la capacité de modélisation mais éliminent certaines possibilités de vérification.

4.4.1 Le principe de la modélisation

Pour déterminer un RPC équivalent à PRS[4] on associe des places du réseau aux nœuds de PRS et des transitions ou des sous-réseaux aux arcs de PRS, comme dans les exemples de la figure 4.7 page suivante. Les déplacements du jeton modélisent l'évolution de l'exécution et sa couleur contient la valeur des variables actuellement définies. Cette dernière règle impose la limitation la plus sévère en termes de ce qu'on peut représenter avec cette méthode: pour que le RPC soit analysable formellement, le nombre de couleurs doit être fini, ce qui implique qu'on ne peut avoir que des variables fermés dans PRS.

À chaque arc on associe deux branches dans le RPC: l'une correspondant au cas où le but est accompli (le jeton va à la place qui correspond au nœud suivant de la KA) et l'autre au cas où le but échoue (la destination du jeton dans ce cas dépend de la position de l'arc dans la KA, comme on le verra par la suite). Pour les buts les plus simples, le test de l'accomplissement du but sera fait par l'étiquette associée à la transition; pour les autres (la majorité), il faut inclure un sous-réseau qui teste l'accomplissement du but (fig. 4.7 page ci-contre).

Une condition nécessaire pour qu'un sous-réseau soit un modèle en RPC d'un arc PRS est qu'il puisse être réduit à un achemineur de jetons. Le réseau consomme le ou les jetons

4. Pour être précis, on devrait dire "équivalent à un sous-ensemble de PRS", ce qu'on ne fera plus dorénavant pour des raisons de concision.

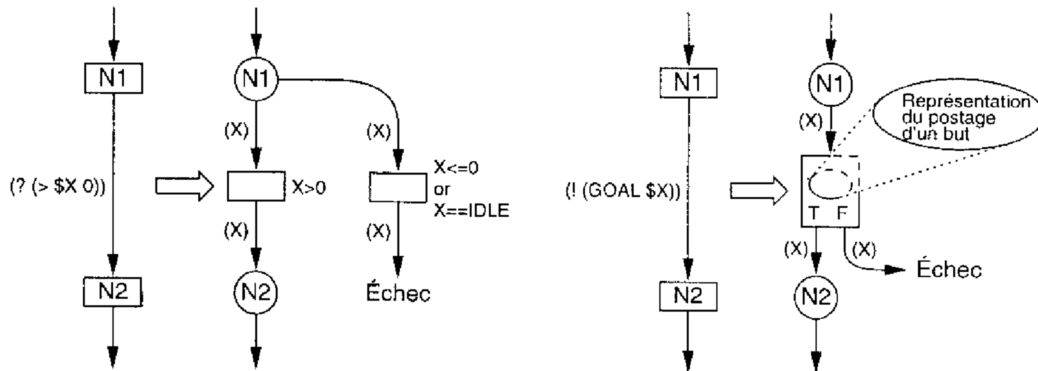


FIG. 4.7 – *Deux exemples d'équivalence entre* PRS *et les RPCs*

présents à son entrée, mais garantit la production d'un ou de plusieurs jetons dans une et dans une seule de ses sorties: celle qui modélise l'accomplissement du but et celle qui modélise son échec (indiquées par T et F respectivement sur nos dessins). D'une manière générale, tout modèle d'un arc doit pouvoir être réduit aux éléments de la figure 4.8, où on garantit que pour tout marquage du réseau il y a au maximum une des deux transitions validée.

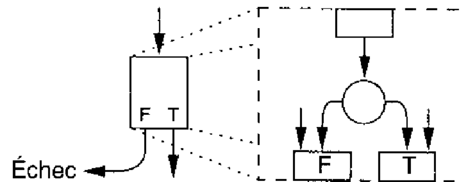


FIG. 4.8 – *Caractéristique essentielle des modèles en RPC des arcs* PRS

4.4.2 La base de données

Les prédicats dans la base de données seront représentés par des places, où l'existence ou non d'un jeton d'une certaine couleur indique si le prédicat a ou n'a pas été établi pour cette valeur de ses arguments. Il y a différents types de prédicats en PRS, chacun d'entre eux correspondant à une représentation différente.

Les prédicats standards

Les prédicats standards vont être représentés, pour chaque prédicat, par quatre places, dénommées AFF DEF, AFF UNDEF, NEG DEF et NEG UNDEF. Les places DEF et UNDEF correspondantes sont complémentaires[5]. La présence d'un jeton d'une certaine couleur dans la

5. Le jeton qui sort d'une place complémentaire doit retourner à la même place ou aller dans l'autre.

place DEF indique que, pour la valeur correspondante, le prédicat a été conclu (affirmativement ou négativement, selon s'il s'agit de la place AFF ou NEG). De façon similaire, un jeton dans la place UNDEF indique que le prédicat n'a pas été conclu pour ces valeurs.

Cette représentation des prédicats standards est redondante, mais nécessaire parce que les réseaux de Pétri ne permettent pas de tester l'absence d'un jeton dans une place. Au lieu de tester son absence, on teste sa présence dans une place complémentaire. Pour que la technique des places complémentaires soit applicable, il faut s'assurer que:

- les places soient bornées, ce qui est toujours vrai pour un nombre fini de couleurs.
- pour chaque paire de places complémentaires, le marquage initial met un jeton de chaque couleur possible dans une des deux places (normalement UNDEF, si on part d'une base de données vide).
- dans le marquage initial du réseau, il n'y a pas de jetons de même couleur simultanément dans les places AFF et NEG.

Si on prend comme exemple un prédicat (EXIST *object colour*), où *object* ∈ {BOOK, CAR}, *colour* ∈ {RED, BLUE, YELLOW}, indéfini au départ pour toutes les combinaisons (*object colour*), la figure 4.9 montre l'état de la représentation du prédicat après les conclusions (=>(EXIST BOOK RED)), (=>(EXIST CAR GREEN)) et (=>(~(EXIST BOOK GREEN))). Cela signifie qu'on est sûr que les livres rouges et les voitures vertes existent, que les livres verts n'existent pas et qu'on ne sait rien à propos des autres combinaisons.

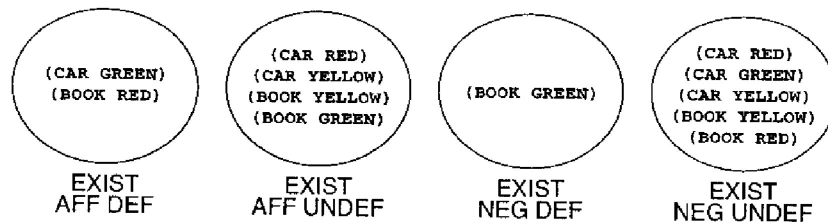


FIG. 4.9 – *Modélisation d'un prédicat standard dans la base de données*

Les tests (? (...)) sur les prédicats standards sont représentés par le RPC de la figure 4.10 page suivante, pour un prédicat générique (PRED *var*) où *var* peut prendre des valeurs de 1 à n. On ne montre le modèle que pour le test de l'affirmation (? (PRED $X)); le modèle du test de la négation (? (~(PRED $X))) est identique, avec les places PRED NEG DEF et PRED NEG UNDEF remplaçant les places PRED AFF DEF et PRED AFF UNDEF, respectivement.

Tout jeton pris dans une des places qui modélisent PRED doit être remis dans la même place, parce qu'un test n'altère pas la valeur d'un prédicat. Le comportement du modèle d'un arc de test est différent (des transitions différentes sont validées) selon l'argument du prédicat:

- L'argument est déjà unifié (a une valeur précise):
  - Si pour cette valeur le prédicat est défini, le test est réussi (transition A).

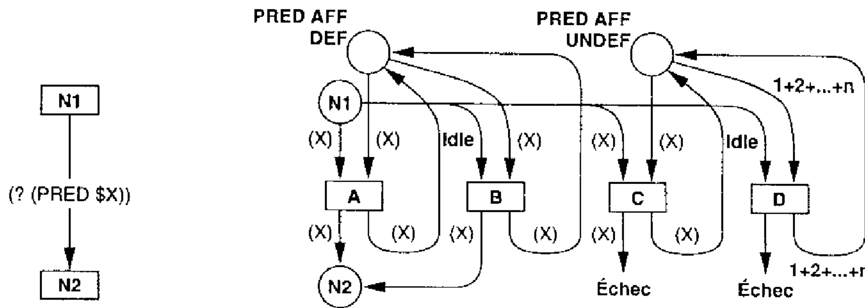


FIG. 4.10 – *Test d'un prédicat standard*

- Si le prédicat est indéfini, le test échoue (transition `C`).

- L'argument n'est pas unifié (`IDLE`):
  - Si on arrive a trouver une valeur pour laquelle le prédicat est défini, le test est réussi et on transmet cette valeur (transition `B`). Ceci représente le mécanisme d'unification de variables.
  - Si `PRED` est indéfini pour toutes les valeurs, le test échoue ((transition `D`).

On montre dans la figure 4.11 le RPC équivalent aux conclusions `(=>(~(PRED $X)))`. Pour la conclusion de l'affirmation, `(=>(PRED $X))`, il suffit d'échanger les places `AFF` et `NEG` correspondantes. On élimine une éventuelle affirmation du prédicat au moment de conclure sa négation (transition `C`). Une conclusion avec des arguments non unifiés, ce qui constitue une erreur, ne change rien à l'état du prédicat (transition `B`).

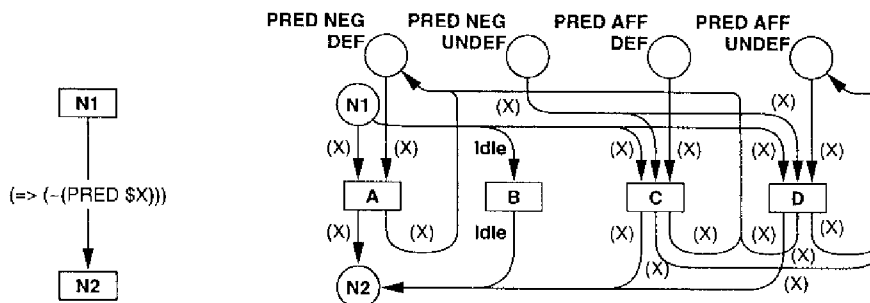


FIG. 4.11 – *Conclusion d'un prédicat standard dans la base de données*

Le modèle des retraits de la base de données `(~>(...))` est montré par la figure 4.12 page suivante. Si l'expression à éliminer n'est pas présente dans la base de données (transition `B`), ou si l'argument est non unifié (transition `C`), le prédicat ne change pas.

Les prédicats fermés

L'absence d'information sur un prédicat standard fait qu'aussi bien le test de l'affirmation que le test de sa négation sont faux. Mais dans plusieurs situations (même la majorité

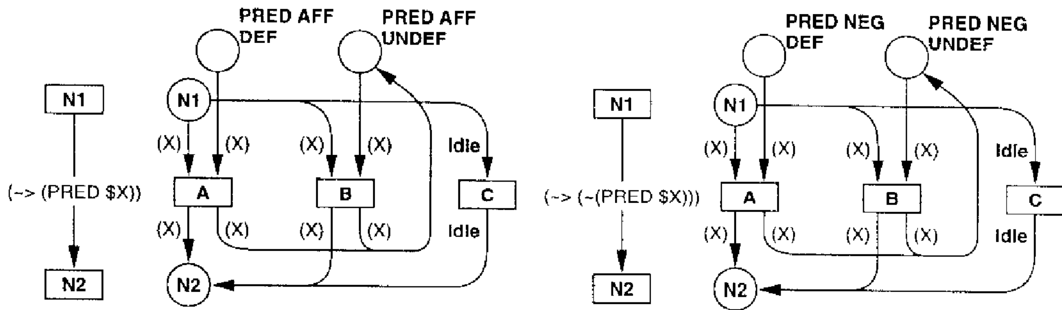


FIG. 4.12 – *Retrait d'un prédicat standard de la base de données*

des cas) on veut que l'absence d'une information soit considérée comme la négation de l'information. Dans PRS, les prédicats déclarés comme fermés présentent ce comportement.

Les prédicats fermés sont représentés par seulement deux places, DEF et UNDEF, étant donné que la non définition du prédicat est indifférenciable de sa négation. Les opérations de retrait de l'affirmation (~> (*exp*)) et de conclusion de la négation (=> ~(*exp*)) sont équivalentes. Le retrait de la négation (~> ~(*exp*)) n'a pas de sens. Dans la figure 4.13 on présente les modèles en RPC des opérations avec les prédicats fermés (seulement pour les affirmations; le modèle pour les négations est identique, avec l'inversion des places DEF et UNDEF).

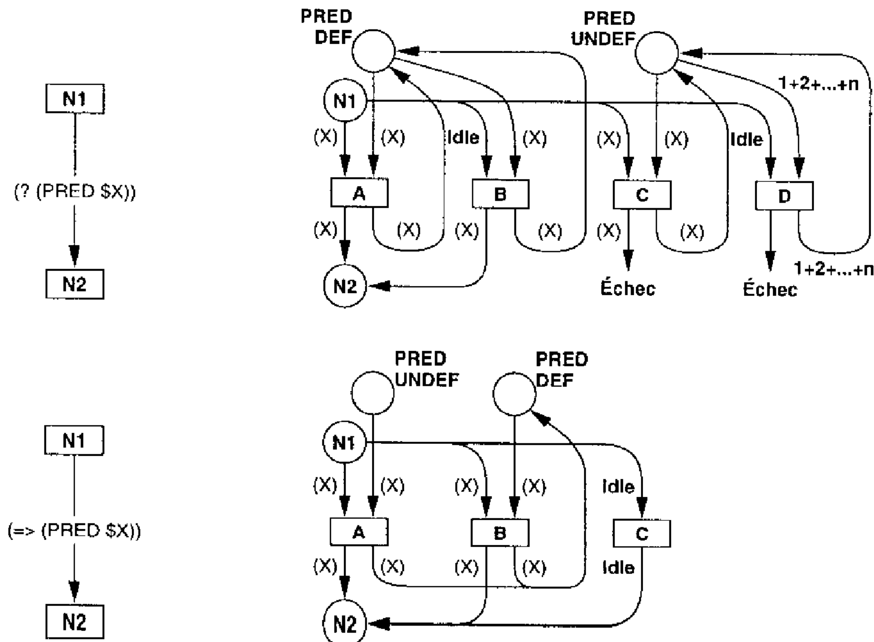


FIG. 4.13 – *Modèles des opérations avec les prédicats fermés*

Les faits fonctionnels

Les faits fonctionnels sont des prédicats où une partie des arguments peut être exprimée comme fonction des autres arguments. Prenons l'exemple d'un prédicat (STATE *module state*): pour un même module, la base de données ne peut contenir qu'un fait avec ce prédicat, étant donné qu'un module ne peut être que dans un seul état à la fois. On peut dire que *state* = STATE(*module*), ce qui caractérise STATE comme un fait fonctionnel.

Les faits fonctionnels en PRS sont des prédicats fermés, mais les représentations en RPC des opérations qui les concernent ont certaines particularités. Par exemple, les jetons dans la place UNDEF ne contiennent pas la partie dépendante des arguments. La figure 4.14, où on suppose que le dernier argument du prédicat n'est pas indépendant, montre les modèles des opérations avec les faits fonctionnels.

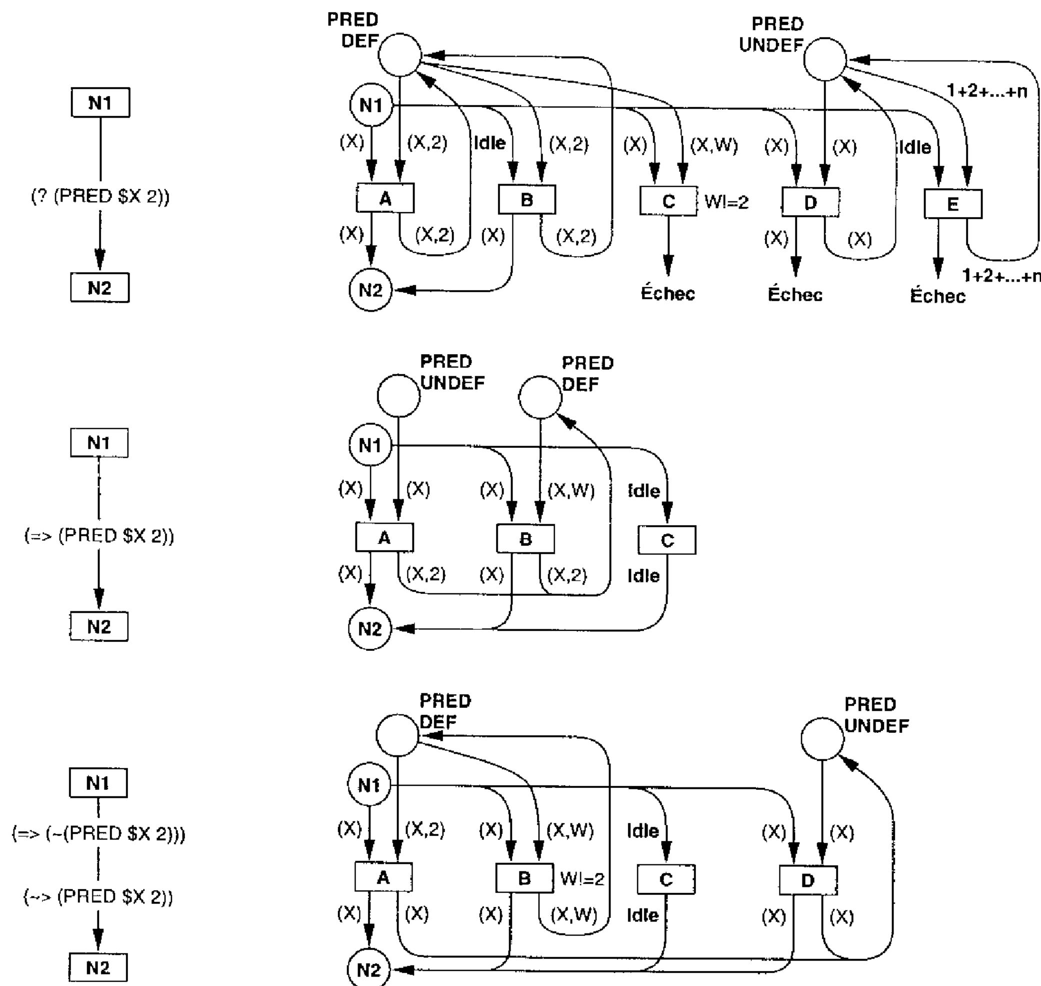


FIG. 4.14 – *Modèles des opérations avec les fait fonctionnels*

Les événements basiques

Les événements basiques sont utilisés pour déclencher des réactions en PRS. Ce sont des faits "volatiles", qui une fois pris en compte disparaissent de la base de données. Ces prédicats sont normalement employés dans les conditions d'invocation des KAs ou pour signaler des événements attendus. Dans la KA de la figure 4.3 page 77, par exemple, les prédicats `ZOMBIE` et `REPLY` sont déclarés comme des événements basiques.

Normalement les événements basiques ne sont pas utilisés dans des tests ou dans des expressions logiques complexes. Ainsi, leur représentation peut se résumer à une simple place, qui servira comme entrée pour les transitions qui attendent cet événement. La transition, une fois déclenchée, peut consommer le jeton, étant donnée la caractéristique fugace de ce type de prédicat.

4.4.3 Les variables

Toutes les variables sont transportées dans des jetons, de leur naissance jusqu'au dernier moment où elles sont utilisées. La figure 4.15 montre un exemple de transmission des variables d'une KA par les jetons d'un RPC. Le modèle des arcs a été simplifié parce que le contexte d'invocation de la KA garantit que les variables `$X` et `$Y` sont unifiées.

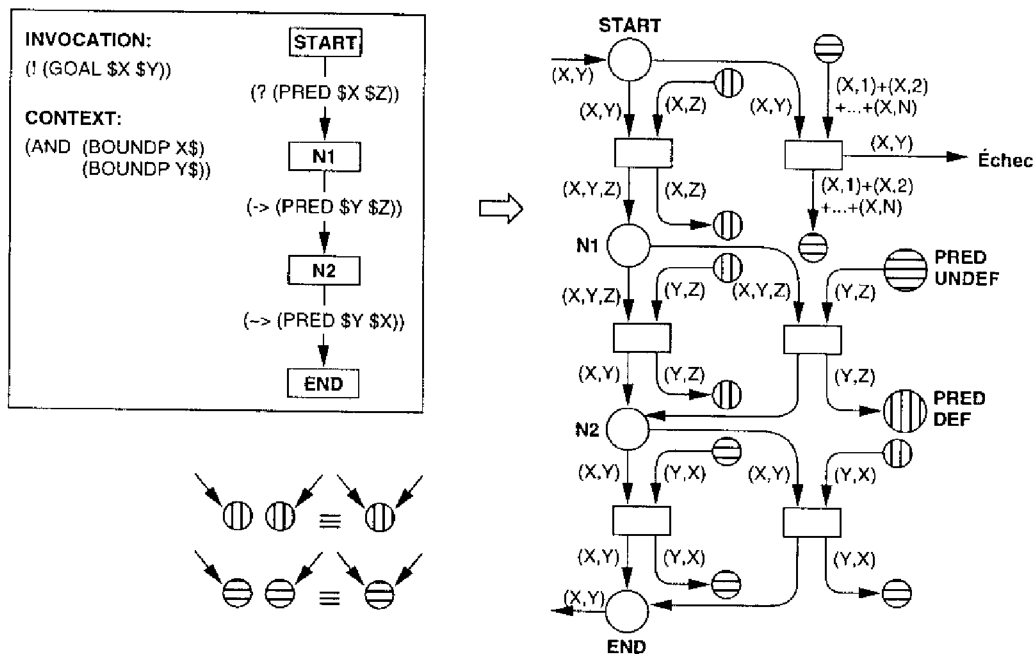


FIG. 4.15 - *La représentation des variables dans les jetons*

Pour les KAs invoquées par un but, les variables qui sont des arguments du but doivent être gardées jusqu'à la fin de la procédure pour établir pour quels arguments le but a ou n'a pas été satisfait. Le jeton sera un n-uplet avec autant d'éléments que de variables PRS

à préserver. Les éléments du jeton auront autant de valeurs possibles que la variable PRS qu'ils représentent, plus un (IDLE) qui indique que la variable n'a pas encore été unifiée.

4.4.4 Les buts

Quand un but est posté, il peut être satisfait soit par consultation de la base de données, soit par un appel à une KA qui va le satisfaire. Le modèle de cette démarche en RPC doit prendre en compte les deux possibilités.

Pour chaque prédicat qui est posté comme un but, on doit avoir, en plus des places qui représentent le prédicat dans la base de données (DEF et UNDEF, en double s'il est un prédicat standard), trois autres places spécifiques aux buts: POST, où la procédure "cliente" signale que le but a été posté, et SUCC et FAIL, où la procédure "serveur" indique le succès ou l'échec dans l'obtention du but. Le placement et la prise de jetons dans ces places représentent le mécanisme d'unification des variables mis en place quand une procédure fait appel à une "sous-procédure".

Dans la figure 4.16 on présente un modèle un peu plus complet du postage d'un but, à être comparé avec le deuxième exemple de la figure 4.7 page 81. Au moment de poster le but, le RPC met un jeton (avec les couleurs des arguments du but) dans la place POST et l'exécution de cette branche entre dans un état d'attente (la place W) d'une réponse quant à l'obtention du but. Cette réponse se matérialisera par l'apparition d'un jeton soit dans la place SUCC, soit dans la place FAIL. Selon la place du jeton, soit l'exécution de la procédure se poursuivra normalement, soit un échec dans la traversé de l'arc est signalé.

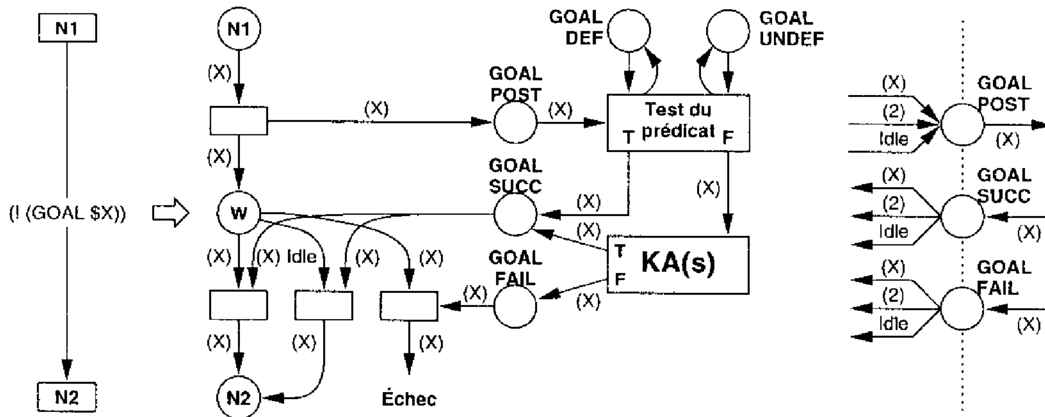


FIG. 4.16 – *Postage d'un but*

Après le postage du but (modélisé par un jeton dans la place POST), le modèle vérifie si le prédicat n'est pas satisfait dans la base de données. Le test est identique à celui effectué en raison d'une demande explicite de test d'un prédicat (figure 4.10 page 83, si le prédicat est standard). Si la base de données contient le prédicat, le but est satisfait; sinon, ce sont les éventuelles KAs déclenchées par ce but qui vont décider de la satisfaction ou non du but (on verra par la suite comment modéliser l'appel aux KAs).

Chaque groupe de places POST, SUCC et FAIL délimite une frontière logique dans le RPC. D'un coté, les procédures "clientes" qui postent des buts et attendent son accomplissement. De l'autre, la ou les KAs qui essayent d'atteindre le but. Dans la figure 4.16 page précédente on n'a représenté qu'une procédure cliente, mais il peut y en avoir plusieurs, comme on le suggère dans le petit dessin à droite de la figure. Chaque KA cliente met un jeton en POST, avec les arguments pour lesquels elle veut que le but soit atteint, et attend l'arrivée d'un jeton de mêmes couleurs dans l'une des places résultat (SUCC ou FAIL).

4.4.5 Les nœuds

Normalement, chaque nœud PRS correspond a une place dans le RPC[6]. Il y a cependant deux types de nœuds qui doivent être représentés d'une manière spéciale: les nœuds condition (IF-THEN-ELSE) et les nœuds qui commencent ou qui terminent une exécution en parallèle.

Les nœuds condition

La traversée d'un arc qui arrive à un nœud condition est toujours possible. Si le but associé à l'arc est accompli, l'exécution se poursuit à partir de la moitié T du nœud condition. Si le but échoue, l'exécution continue de la moitié F. Il est donc naturel qu'on représente ce nœud par deux places dans le RPC: une qui modélise la moitié T et l'autre la moitié F du nœud. Chaque modèle d'un arc doit générer deux sorties possibles pour le jeton: on va diriger la branche qui représente l'accomplissement du but vers la place T et la branche qui modélise l'échec vers la place F, comme on l'indique dans la figure 4.17.

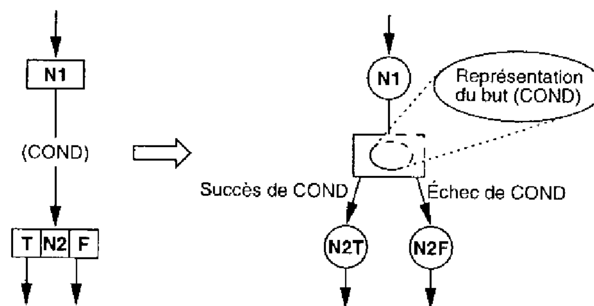


FIG. 4.17 – *Modèle d'un nœud condition*

Le parallélisme

C-PRS permet l'exécution de plus d'une branche en parallèle dans une KA. Un nœud de partage (indiqué par une barre en bas du rectangle) lance autant de branches d'exécution en parallèle que d'arcs qui partent de lui, alors qu'un nœud de jonction (indiqué par une

6. Comme il y a beaucoup plus de places que de nœuds, la réciproque n'est pas vraie.

barre en haut du rectangle) ne poursuivra l'exécution qu'après l'accomplissement des buts de tous les arcs arrivant en lui. L'échec d'une des branches signifie l'échec de la KA entière.

Le modèle pour ces nœuds, présenté dans la figure 4.18, fait apparaître, en plus de la place normalement associée au nœud, une transition et autant de places supplémentaires que de branches exécutées en parallèle. Pour un nœud de partage, une fois qu'un jeton arrive à la place correspondante, la transition permet l'exécution en parallèle de toutes les branches, mettant un jeton dans chacune de leurs places de départ. Le nœud de jonction, à l'inverse, fait apparaître une transition qui n'est validée qu'une fois que les jetons de toutes les branches sont arrivés. À ce moment, tous les jetons des branches en parallèle sont consommés et un jeton est mis dans la place qui représente le nœud, pour permettre la pursuite de l'exécution.

FIG. 4.18 – *Modèles des nœuds d'ouverture et de fermeture du parallélisme*

4.4.6 Les arcs

Les tests

On a déjà présenté des arcs de test (? *COND*) dans des situations où la condition à tester porte sur des variables (premier exemple de la figure 4.7 page 81) et sur des prédicats (figures 4.10 page 83, 4.13 page 84 et 4.14 page 85). Ces deux types de test peuvent être combinés dans un même arc, ce qui va générer un modèle de représentation plus complexe. Les fonctions (+, /, ...) et les prédicats (<, ==, ...) arithmétiques évaluables prédéfinis peuvent aussi faire partie des expressions.

Pour illustrer ceci, supposons que les variables X et Y ont un univers de variation de 0, 1 ou 2, que le prédicat PRED est fermé et que l'opération d'addition est circulaire ($2+1=0$), pour rester dans le domaine de variation possible, la figure 4.19 page suivante montre le modèle assez complexe en RPC d'un arc qui teste la condition

```
(AND  (PRED (+ $X 1) $Y)
      (~(== $X $Y)) )
```

Les tests de prédicats sont les arcs qui requièrent les RPCs les plus complexes pour les modéliser. Quelques règles peuvent être adoptées pour simplifier les modèles:

- Les modèles génériques des tests sur des prédicats avec plus d'un argument (comme

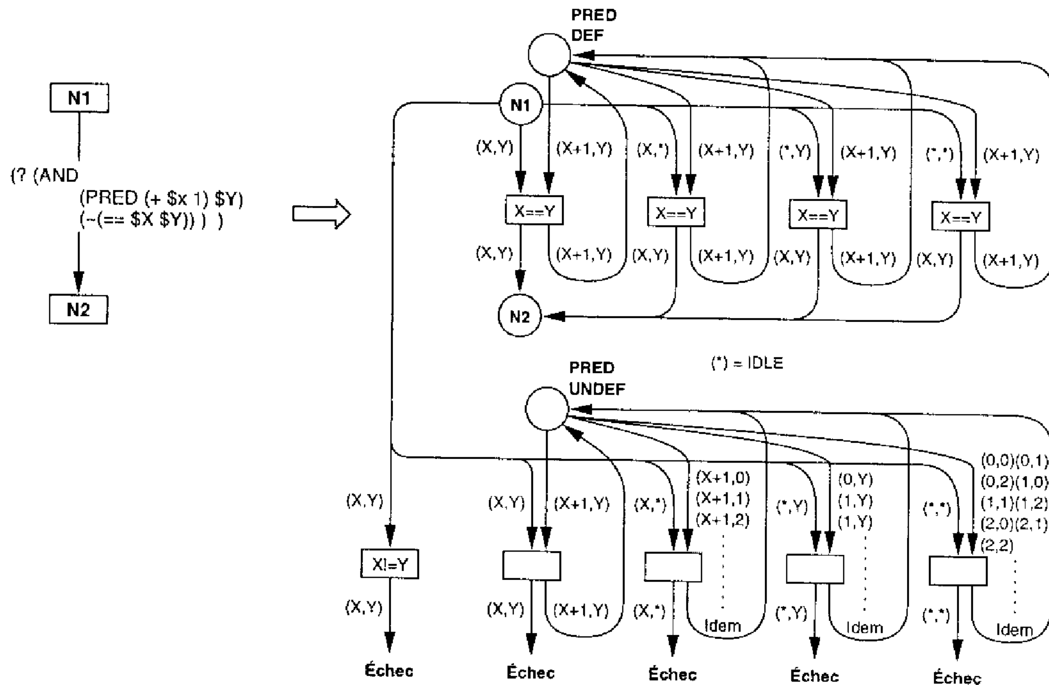


FIG. 4.19 – *Modèle d'un arc qui teste une condition complexe*

celui de la figure 4.19) sont très complexes parce qu'il faut prévoir toutes les combinaisons possibles entre variables unifiées et non unifiées. On peut réduire la taille du RPC par:

- réduction du nombre d'arguments des paramètres; ou
- déclaration des variables unifiées dans une KA (les variables d'entrée), comme dans l'exemple de la figure 4.15 page 86, où l'utilisation du prédicat standard `BOUNDP` garantit que la KA ne peut être exécutée que si les variables d'entrée ont une valeur définie; cela génère des modèles simples pour les tests.

- Les prédicats standards sont plus complexes que les prédicats fermés. Ces derniers peuvent remplacer les premiers dans la plupart des situations, et sont même plus proches de la façon humaine habituelle de raisonner.
- Théoriquement, rien ne nous empêche de combiner plusieurs tests dans un seul arc avec des opérateurs logiques. Cela peut cependant conduire à des modèles trop complexes, qui peuvent être évités dans certains cas par substitution des `AND` par des arcs en série et dans plusieurs cas par substitution des `OR` par des arcs en parallèle.

Les demandes d'accomplissement

Les demandes d'accomplissement `(! GOAL)` peuvent être utilisées avec des prédicats définis par l'utilisateur ou avec des primitives.

Pour les prédicats définis par l'utilisateur, le modèle de la procédure de postage d'un but a été montré dans la figure 4.16 page 87. Il reste à expliquer le mécanisme de choix des KAs pour accomplir les buts qui ne sont pas satisfaits dans la base de données, ce qui sera fait dans la section 4.4.9.

La plupart des primitives utilisées dans les demandes d'accomplissement correspondent à des KAs primitives dont les actions n'échouent jamais (PRINTF, ...), et qui n'ont pas d'intérêt pour l'analyse. Ces arcs peuvent être remplacés par des transitions toujours satisfaites. Une exception notable est la primitive d'affectation (=): si l'affectation est toujours possible pour les variables de programme, elle ne l'est pour les variables logiques que si elles ne sont pas encore unifiées (figure 4.20).

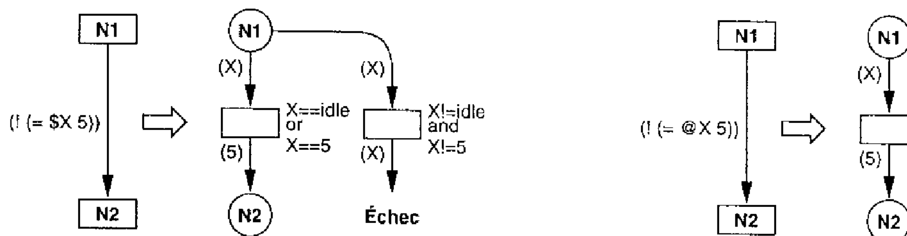


FIG. 4.20 – *Modèle de l'affectation d'une variable*

Les attentes

Un arc d'attente n'échoue jamais: il peut à la limite rester en attente éternellement. Son modèle ne prévoit pas d'issue en cas d'échec, mais seulement une transition qui sera franchie quand la condition attendue se vérifiera (fig. 4.21).

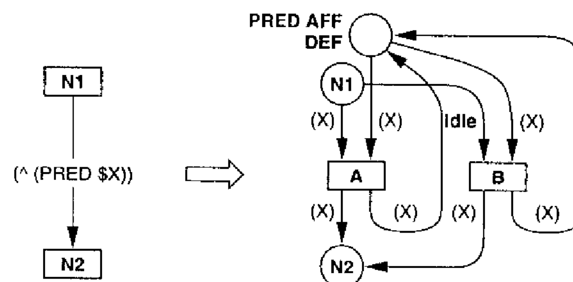


FIG. 4.21 - *Modèle d'un arc d'attente*

4.4.7 Le non déterminisme

En PRS, quand plusieurs arcs partent d'un même nœud, le système choisit un des arcs au hasard pour essayer de le traverser. S'il ne réussit pas, il choisit un autre et ne considère que la KA a échoué qu'après avoir essayé tous les arcs. Il faut signaler que PRS ne fait

pas de retour en arrière: s'il réussit à traverser un arc, mais échoue dans la suite de cette branche, il ne revient pas aux nœuds antérieurs pour tester les arcs qu'il n'a pas exploré.

Si le choix non déterministe est facile à modéliser avec les réseaux de Pétri (il suffit de mettre des transitions en conflit), il faut concevoir une structure spécifique pour garantir que le modèle va essayer de parcourir toutes les branches avant de signaler une erreur. Cette structure est présentée dans la figure 4.22 pour un cas avec deux arcs en conflit, mais est facilement adaptable à n'importe quel nombre d'arcs.

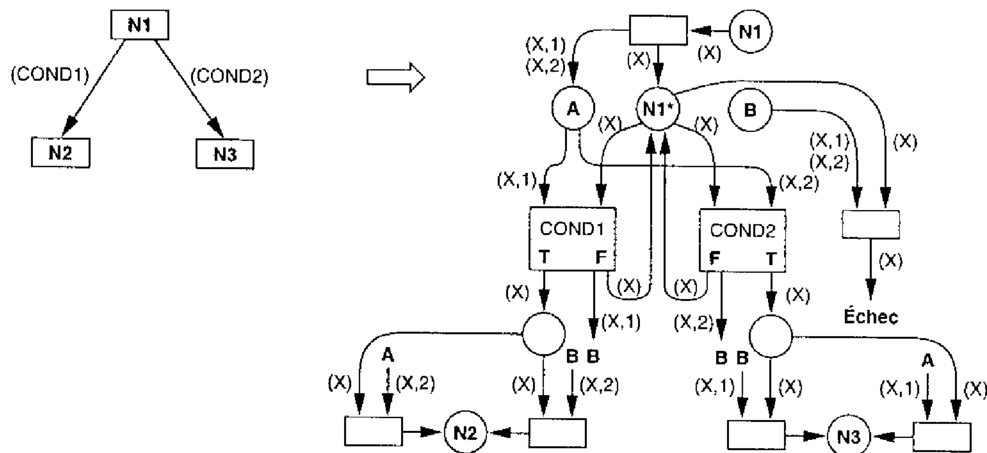


FIG. 4.22 – *Nœud avec plusieurs arcs partants*

La place A, juste après le tir de la transition dont N1 est l'entrée, contient n (2 en l'occurrence) jetons numérotés, un pour chaque branche. La branche ne peut s'exécuter que si son jeton est encore en A et si le jeton de N1* est disponible (les transitions en conflit pour le jeton de N1* modélisent le choix non déterministe). Si l'exécution de la branche qui prend le jeton de N1* est réussie, elle nettoie les places A et B (consomme tous les jetons des autres branches) et l'exécution se poursuit. Si elle échoue, le modèle remet le jeton en N1*, pour qu'une autre branche puisse s'exécuter, et met le jeton correspondant à la branche qui a échoué en B, place qui maintient la mémoire des tentatives déjà réalisées. Si jamais tous les jetons numérotés se retrouvent dans B, le modèle signale un échec.

4.4.8 Les KAs

Tout RPC qui modélise une KA a une place de début, qu'on appelle INIT. Entre INIT et START se trouve un sous-réseau qui teste si le contexte d'exécution (le champ CONTEXT) de la KA est satisfait. Cela utilise le fait que, du point de vue logique et si on oublie des considérations liées à la mise en œuvre, il est équivalent d'avoir une KA avec un contexte d'exécution et la même KA, sans le contexte d'exécution, mais avec un arc supplémentaire avant l'ancien nœud START et qui teste les mêmes conditions que le champ CONTEXT (fig. 4.23 page suivante). Pour les KAs qui n'exigent aucun contexte d'exécution, la place INIT se confond avec la place START.

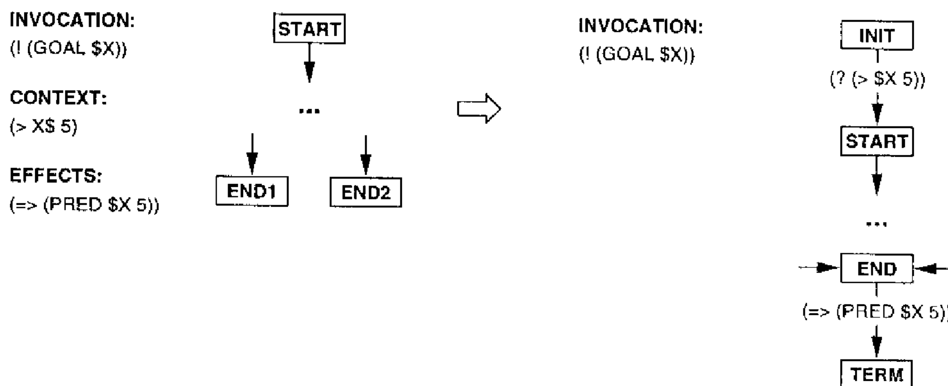


FIG. 4.23 – *Équivalence entre certains champs d'une KA et des arcs supplémentaires*

Le modèle d'une KA a une seule place `END`, qui joue le rôle de tous les nœuds terminaux de la KA. On ajoute, après cette place, le modèle des conclusions et retraits de la base de données prévus dans le champ `EFFECTS`, comme s'ils avaient été effectués par un arc supplémentaire après la place `END` (fig. 4.23), et une place `TERM` qui modélise la terminaison réussie de l'exécution de la KA. Pour les KAs qui n'ont pas de champ `EFFECTS`, la place `TERM` se confond avec la place `END`.

Exceptées les rares KAs qui n'échouent jamais, les RPCs doivent aussi contenir une place `FAIL`, où l'apparition d'un jeton indique que l'exécution de la KA a échoué. L'issue en cas d'erreur (F) de tous les modèles des arcs est de placer un jeton dans la place `FAIL`, sauf pour les cas spéciaux que l'on a déjà présenté:

- L'arc conduit à un nœud `IF-THEN-ELSE`. Dans ce cas, le jeton en cas d'échec doit être mis dans la place qui représente la moitie `F` du nœud condition (section 4.4.5).
- L'arc part d'un nœud avec plusieurs arcs partants. Le mécanisme de choix non déterministe d'un autre arc à traverser doit alors être utilisé (section 4.4.7).

Dans la figure 4.24 page suivante on montre un exemple du RPC équivalent à une KA générique. Les modèles des arcs proprement dits ne sont pas détaillés, et on suppose qu'il n'y a qu'une seule variable à être transmise (`X`) tout au long de la KA. Si on fait l'analyse (par génération du graphe d'accessibilité) de ce réseau, en remplaçant les modèles génériques des arcs par le sous-réseau de la figure 4.8 page 81, on constate, comme il se doit pour tous les modèles des KAs, que le placement d'un jeton dans `INIT` mène a l'apparition du même jeton, soit dans `TERM`, soit dans `FAIL`, pour tous les scénarios d'exécution envisageables.

4.4.9 L'activation des KAs

Activer une KA pour exécution, dans notre modèle, correspond à mettre un jeton dans sa place `INIT`. Cet événement ne peut se produire que si le facteur de déclenchement de la KA a lieu: une modification dans la base de données ou le postage d'un but.

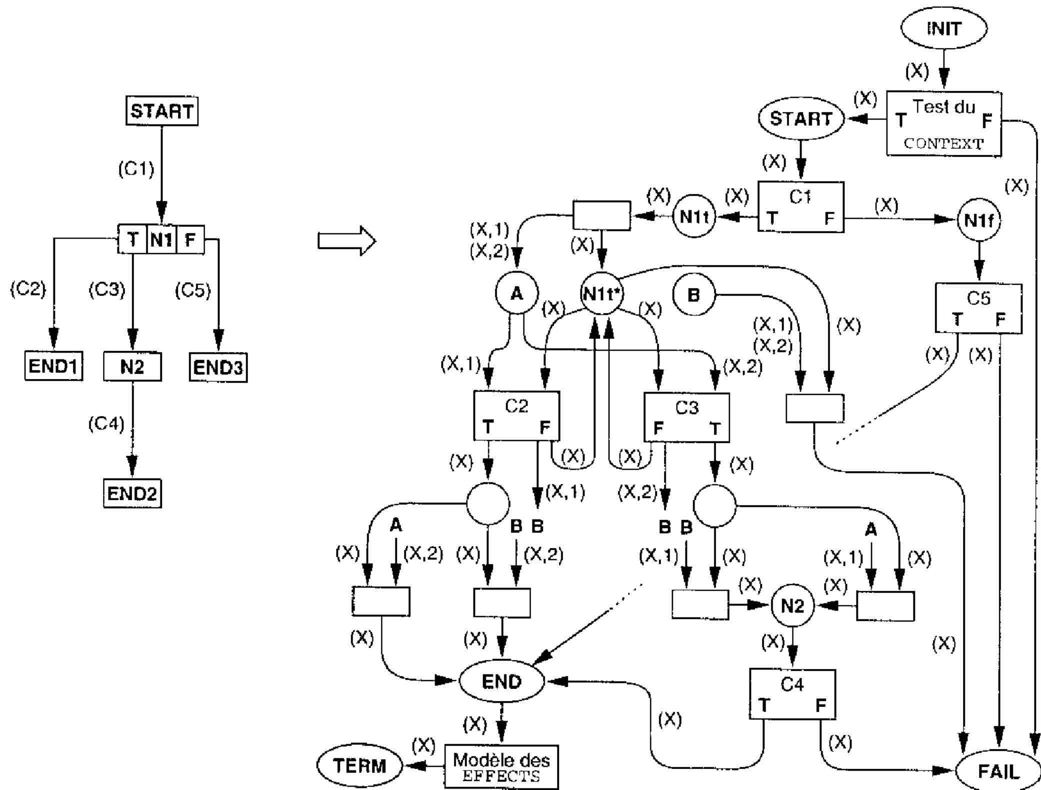


FIG. 4.24 – *Exemple de réseau de Pétri coloré équivalent à une KA*

Les KAs déclenchées par des faits

Pour les KAs qui réagissent à des faits, une transition détecte l'apparition du fait et place un jeton dans `INIT`. Si le prédicat en question est un événement basique et si cette KA est la seule activée par ce prédicat, la transition consomme le jeton (fig. 4.25 page suivante); sinon, le jeton est remis dans sa place originale. Les fins d'exécution des KAs qui ne sont pas associées à la satisfaction de buts n'ont pas de conséquence, de sorte que des transitions sans place de sortie consomment simplement les jetons qui apparaissent dans les places `TERM` et `FAIL` du modèle de ces KAs (fig. 4.25 page ci-contre).

Les KAs déclenchées par des buts

Pour les KAs déclenchées par des buts, il faut revenir au modèle du postage d'un but (fig. 4.16 page 87). On va distinguer trois cas:

1. Aucune KA n'est déclenchée par le but.
2. Une seule KA est déclenchée par le but.
3. Plusieurs KAs sont déclenchées par le but.

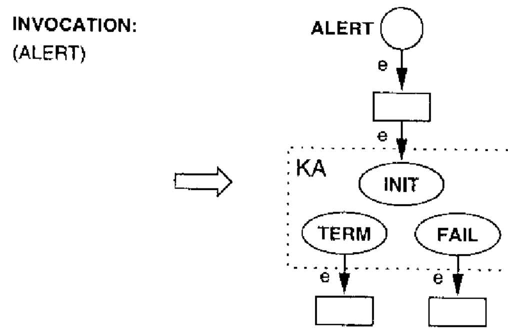


FIG. 4.25 – *Déclenchement d'une KA activée par un fait*

Si aucune KA n'est déclenchée par le but, le test de son accomplissement se résume au test de la présence du prédicat dans la base de données (fig. 4.26).

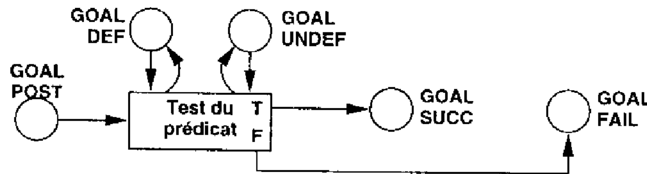


FIG. 4.26 – *But pour lequel aucune KA n'est déclenchée*

Quand il n'y a qu'une seule KA qui satisfasse le but, et si celui-ci n'a pas pu être accompli dans la base de données, son succès ou son échec est indissociable du succès ou échec de la KA (fig. 4.26).

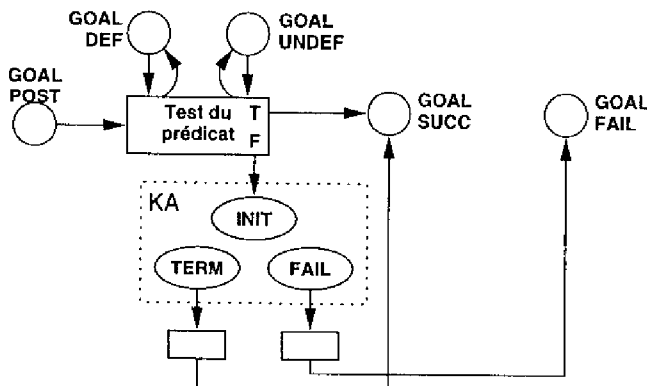


FIG. 4.27 – *But pour lequel une seule KA peut être déclenchée*

Finalement, quand il y a plusieurs KAs déclenchées par le même but, et en absence d'une meta-KA qui choisisse une d'entre elles, PRS va faire le choix de façon non déterministe. Le problème est similaire au choix d'un arc entre plusieurs qui partent d'un même

nœud (section 4.4.7), étant donné que le système doit essayer toutes les KAs avant de pouvoir signaler une erreur. On va utiliser une structure similaire à celle de la figure 4.22 pour garantir que toutes les KAs seront essayées, ce qui mène, pour un exemple avec deux KAs, au modèle de la figure 4.28.

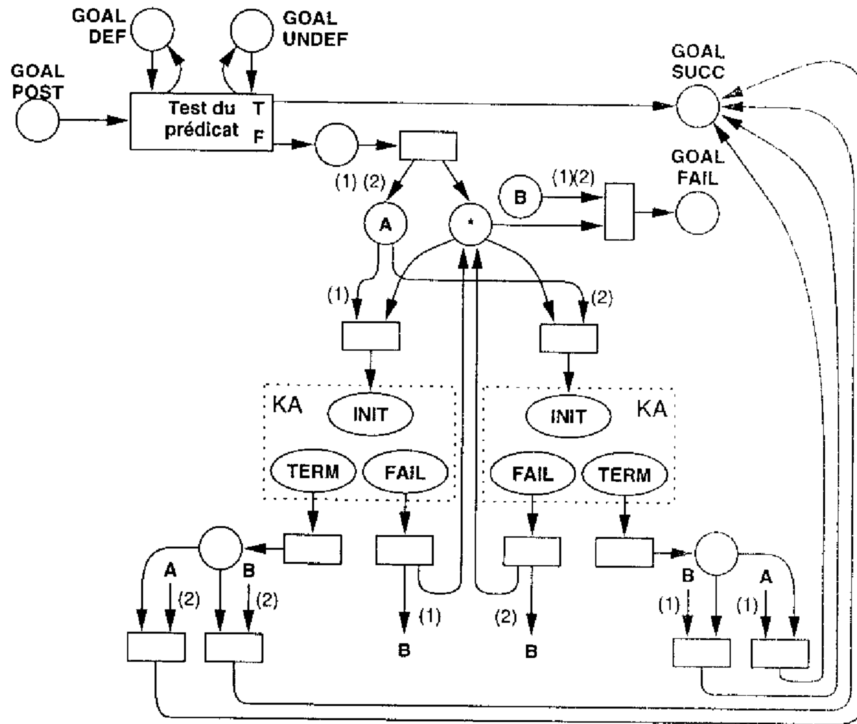


FIG. 4.28 – *But pour lequel plusieurs KAs peuvent être déclenchées*

4.4.10 Les limitations

La plupart des limitations de cette méthode de représentation de PRS sont implicites dans les divers modèles présentés. On liste ci-après quelques unes des plus notables.

Préservation automatique d'un attribut

On n'a pas obtenu d'équivalents généraux pour les opérateurs de préservation passive (`# C`) et active (`% C`) d'un prédicat. Ces opérateurs ne sont jamais utilisés tous seuls, mais dans des conjonctions du type `(& (! (GOAL $X)) (# (PRED $Y)))`. Une possibilité pour les représenter serait d'ajouter à tous les arcs de la KA déclenchée par le but `GOAL` le test de la condition `PRED` à préserver. Les modèles obtenus seraient cependant trop complexes, ce qui nous a fait préférer de les considérer comme sans équivalents directs. Un sous-ensemble de PRS sans ces opérateurs ne perd pas beaucoup de sa puissance, étant

donné que certains comportements similaires peuvent être obtenus avec des KAs activées par des faits.

Domaine fermé des variables

Cette limitation exclue l'utilisation des réels et de toute autre donnée non dénombrable. Les entiers sont possibles si on impose des limites. L'impossibilité d'utiliser les réels, entre autres conséquences, interdit de faire intervenir le temps dans les tests, sauf s'il est discrétisé.

Les meta-KAs

Comme les meta-KAs manipulent des données complexes, elles sont très difficilement représentables. Des meta-KAs particulières peuvent cependant être incorporées au RPC de façon individualisée. Par exemple, des critères de sélection de KAs autres que le non déterminisme pur, si on peut les modéliser, peuvent substituer la structure de la section 4.4.9.

4.5 Vérification du niveau tâche

On ne va pas détailler dans ce mémoire les méthodes de vérification des RPCs, qui sont d'ailleurs bien présentées dans la littérature, principalement dans les travaux de Jensen [Jensen 90, Jensen 94]. Très synthétiquement, on peut dire que les RPCs peuvent être analysés de trois manières principales:

Simulation – La simulation d'un RPC est très similaire à l'exécution segmentée d'un programme pour éliminer des erreurs (*debugging*). Différents outils graphiques permettent de fixer des points d'arrêt d'exécution (*breakpoints*), exhiber le marquage courant, calculer des statistiques à propos du système modélisé, etc.

Graphe d'accessibilité – L'idée des graphes d'accessibilité (aussi connus comme espaces d'états ou graphes d'occurrence) est de construire un graphe orienté qui a un nœud pour chaque état atteignable du réseau et un arc pour chaque changement d'état. Évidemment, un tel graphe peut devenir très grand, même pour des petits réseaux. Cependant, il peut être construit et analysé de façon totalement automatique. Il existe en outre des techniques avec lesquelles il devient possible de travailler avec des graphes d'accessibilité condensés, sans perdre le pouvoir analytique.

Invariants – La méthodes des invariants (de place ou de transition) est très similaire à l'utilisation d'invariantes dans la vérification de programmes ordinaires. La technique consiste à construire un ensemble d'équations qui sont prouvées être satisfaites dans tous les états du système. Les équations sont alors utilisées pour prouver certaines propriétés du système modélisé, comme l'absence de blocages (*deadlocks*).

Comme les RPCs sont hiérarchiques, on peut définir des sous-réseaux qui sont intégrés au réseau principal, comme des sous-routines dans un programme. L'analyse peut partir des réseaux élémentaires vers le réseau principal, ce qui réduit la complexité du problème.

Les modèles des arcs PRS sont des candidats naturels à devenir des sous-réseaux du modèle global, comme on l'a déjà intuitivement fait dans les schémas des exemples de la section précédente.

Les méthodes d'analyse formelle permettent de prouver un certain nombre de propriétés pour les réseaux de Pétri. Ces propriétés peuvent être soit comportementales (pour un certain marquage initial) soit structurelles (qui ne dépendent que de la structure du réseau). Les invariants établissent certaines propriétés structurelles, alors que le graphe d'accessibilité permet, par inspection des états, de déterminer toutes les propriétés du réseau, mais seulement pour un marquage initial bien précis.

Pour le RPC de notre modèle, les propriétés qui nous intéressent plus particulièrement sont celles prouvées pour un marquage initial défini, pour deux raisons:

- On utilise des places complémentaires dans le réseau. Ces places exigent que l'on garantisse qu'à tout instant (y compris l'instant initial) tous les jetons possibles soient dans l'une des places complémentaires, ce qui ne donne un sens au système que si on fixe un marquage initial.
- Normalement, ce qui nous intéresse est de pouvoir vérifier la réaction du robot face à des situations (des marquages) précises, qui correspondent au postage du (ou des) but(s) de plus haut niveau que l'on a prévu pour le système.

Les propriétés démontrées pour le RPC doivent être interprétées sachant que le réseau modélise un ensemble de KAs et à partir de ce que le marquage initial représente. Une place non bornée en nombre de jetons, par exemple, indique presque certainement une erreur de conception des KAs (comme une KA qui pourrait s'appeler récursivement sans limites). L'existence de blocages, cependant, peut être soit la preuve d'une erreur de conception, soit une condition nécessaire pour que le système puisse être considéré comme correct:

- Si le comportement que l'on attend du robot, pour un certain marquage initial, est qu'il rentre dans un cycle sans fin, l'existence d'une possibilité de blocage indique une erreur de conception.
- Pour un marquage initial où il y a un but posté, il faut nécessairement que le modèle se mette finalement en blocage, avec un jeton dans une des places qui indiquent le succès ou l'échec du but posté.

Les possibilités de vérification offertes par la modélisation du niveau tâche n'existent que si le modèle obtenu est représentatif de l'ensemble de procédures PRS. Un tel modèle ne peut être obtenu que dans le contexte suivant:

- Un exécutif, comme celui décrit dans le chapitre 3, prend en charge la manipulation des données non dénombrables.
- Les procédures PRS n'utilisent pas les caractéristiques non représentables du système (comme celles de la section 4.4.10).

Dans le contexte spécifique de la mise en œuvre du niveau tâche de notre architecture de contrôle, ces restrictions peuvent être respectées. D'une manière générale, cependant, la

possibilité de vérification d'un ensemble de procédures PRS par les RPCs dépend fortement du type d'utilisation qu'on fait du système PRS.

Pour les activités externes au niveau tâche, la vérification des KAs peut être faite dans un premier temps en supposant que tous les arcs qui font des appels aux services de l'exécutif réussissent. Cela permet de tester la correction intrinsèque du niveau tâche et de faire une certaine validation de l'ensemble de procédures (vérifier qu'elles mettent bien en œuvre le comportement souhaité). Ensuite, les appels au niveau fonctionnel peuvent être remplacés (tous à la fois où un par un) par des sous-réseaux qui peuvent aussi échouer, ce qui permet de tester les éventuelles KAs de reprise et le comportement dégradé du système.

4.6 Conclusion

Dans ce chapitre, nous avons présenté des procédures de communication avec l'exécutif qui permettent à PRS d'utiliser les services du niveau fonctionnel dans la mise en œuvre du niveau tâche. Ensuite, un sous-ensemble de PRS pour lequel une équivalence avec les réseaux de Pétri colorés existe est défini, ce qui permet une vérification formelle partielle du niveau tâche.

Le chapitre suivant montre une mise en œuvre de l'architecture, simplifiée dans les aspects externes au contrôle pour permettre d'illustrer de façon plus complète la gestion du système. L'exemple inclut l'exécutif, qui contrôle un ensemble de modules, et un niveau tâche, écrit en PRS et vérifié en suite par équivalence avec les réseaux de Pétri colorés.

Chapitre 5

Expérimentations

Nous avons décrit précédemment certains principes d'une architecture de contrôle générale pour robots mobiles autonomes. Dans ce chapitre nous montrons les résultats qu'offre leur mise en œuvre dans un environnement réel. Deux objectifs principaux sont visés avec l'expérimentation:

- Montrer que l'exécutif peut surveiller un ensemble de modules en respectant les contraintes temporelles qu'on lui a imposées.
- Vérifier que la stratégie de distribution des données et de résolution de conflits entre services permet au niveau tâche d'accomplir ses objectifs avec une possibilité de vérification formelle.

5.1 Description de l'expérimentation

L'exemple que nous avons retenu concerne un robot d'intérieur, qui explore un environnement à la recherche d'un certain type d'objet. Dès qu'un objet est détecté, le robot doit se diriger vers lui, le prendre et le déposer sur le robot lui-même, jusqu'à ce que le nombre désiré d'objets (un paramètre connu d'avance) de ce type ait été trouvée. Une première procédure simple pour traiter ce problème est la suivante:

Jusqu'à ce que la totalité des objets ait été trouvée:
- Effectuer une recherche.
- Actualiser le modèle de l'environnement.
- **Si** un objet a été trouvé:
  - Calculer une trajectoire qui mène à l'objet.
  - Faire le déplacement.
  - Prendre l'objet et le déposer sur le robot.
- Déterminer une nouvelle position d'exploration.
- Calculer une trajectoire vers la position d'exploration.
- Faire le déplacement.

Fin de l'expérimentation

Cette approche, même si elle peut donner de bons résultats, sous-utilise les capacités du système en raison de la nature séquentielle des opérations: le robot pourrait par exemple faire des recherches pendant les déplacements ou planifier la trajectoire suivante pendant la prise d'objet. Ces possibilités peuvent être concrétisées avec un algorithme comme celui de la figure 5.1, où trois cycles s'exécutent en parallèle:

1. **Prise d'objet:** quand un objet est repéré, le cycle de prise d'objet calcule une trajectoire de la position courante du robot jusqu'à la position de l'objet et l'exécute. Une fois arrivé, le robot prend l'objet (avec un bras manipulateur, par exemple) et déposé sur lui-même, après quoi le cycle se met de nouveau en attente.
2. **Exploration:** ce cycle, à son tour, détermine sans interruption de nouvelles positions pour observer l'environnement, sur la base d'un modèle qui permet de calculer les régions peu explorées, où le robot a plus de chances de trouver de nouveaux objets.
3. **Modélisation:** ce cycle acquiert en permanence des images autour du robot et cherche des objets dans les images (ce qui actualise le modèle de l'environnement).

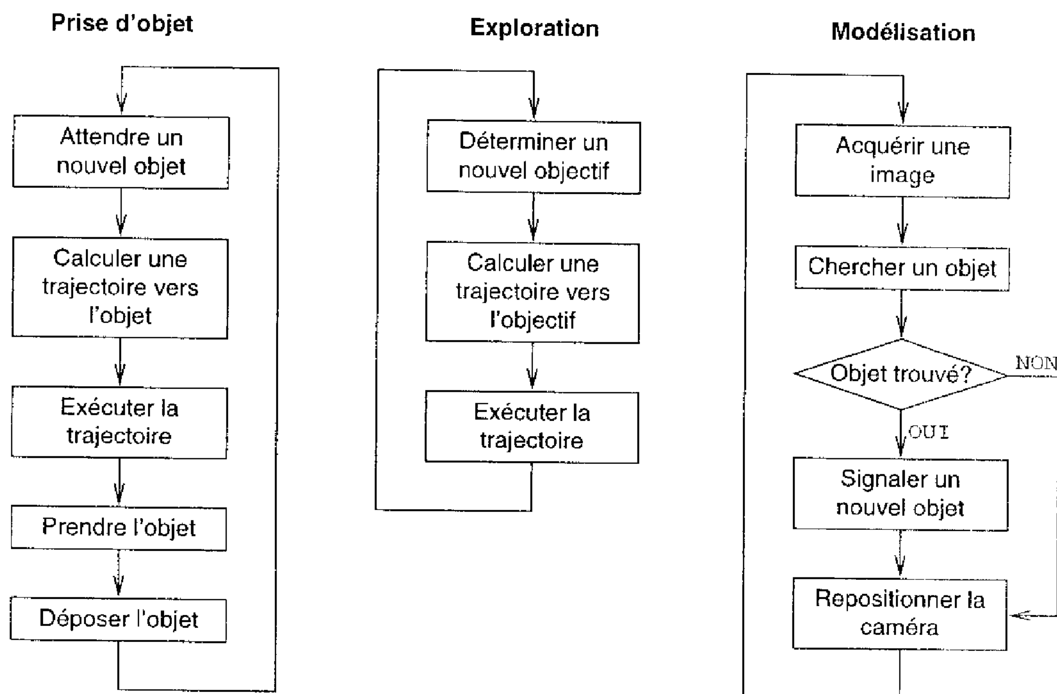


FIG. 5.1 – *Un algorithme parallèle pour l'expérimentation*

La parallélisation des activités permet d'optimiser l'utilisation des ressources du robot mais peut créer des situations de conflit, qui doivent absolument être prises en compte:

- l'exécution simultanée de deux trajectoires est évidemment impossible; et
- la prise d'un objet (mais pas la pose) dans cet exemple exige l'immobilité du robot.

Ces conflits doivent être arbitrés en fonction de la tâche globale du robot. Comme l'objectif ultime de la mission est la prise d'objets, les activités de ce cycle doivent être considérées comme prioritaires par rapport à celles des deux autres. Ainsi, l'exécution de trajectoire et la prise d'objet dans ce cycle vont interrompre ou faire attendre l'exécution de la trajectoire du cycle d'exploration.

On peut aussi introduire intentionnellement d'autres incompatibilités qui, si elles ne se sont pas justifiées par un conflit d'utilisation d'une ressource physique du robot, augmentent l'efficacité du système:

- Pendant les déplacements, le robot continue à actualiser son modèle de l'environnement. Comme les exécutions de trajectoire du cycle d'exploration attendent la fin des mouvements du cycle de prise d'objet, il est préférable que la détermination d'un nouvel objectif attende elle aussi la fin du déplacement, pour utiliser le modèle le plus récent au moment où la trajectoire d'exploration pourra être exécutée.
  Pour la même raison, le temps d'attente maximal (*timeout*) de l'exécution de la trajectoire d'exploration doit être petit: si l'on ne peut l'exécuter que plus tard, il est alors préférable d'avoir un objectif et en conséquence une trajectoire basés sur un modèle plus actualisé à ce moment là.
- Dès qu'un objet est détecté, l'éventuelle exécution de la trajectoire d'exploration doit s'arrêter, étant donné que la poursuite de ce déplacement peut le faire sortir de la zone de visibilité. On introduit donc une incompatibilité entre le calcul de la trajectoire vers l'objet trouvé et l'exécution de la trajectoire d'exploration.

Les cycles vont correspondre à des procédures PRS au niveau tâche et les incompatibilités entre services seront mises en place dans l'exécutif, qui utilise les fonctions d'un groupe de modules (fig. 5.2). Certains modules offrent des services au niveau décisionnel (à travers l'exécutif), alors que les autres ne sont utilisées que comme des serveurs pour les modules du premier groupe.

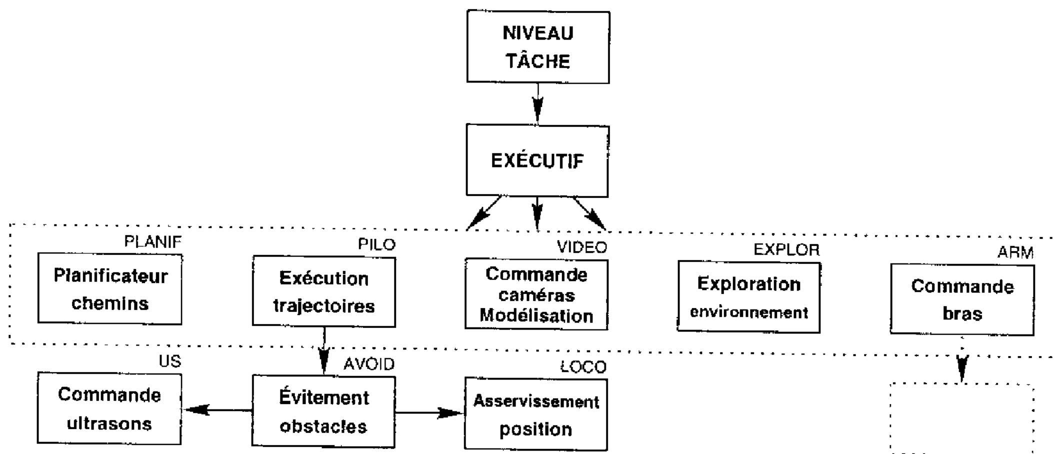


FIG. 5.2 – *L'architecture de l'expérimentation*

5.2 Les modules mis en œuvre

L'expérimentation est basée sur le robot *Hilare* 2, dont l'ensemble des modules est présenté dans la section 2.3 page 35. Nous avons utilisé les modules de base suivants:

- `PLAN` – planificateur de trajectoires.
- `PILO` – génération de trajectoire;
- `AVOID` – évitement d'obstacles;
- `US` – les ultrasons; et
- `LOCO` – asservissement en position;

Les autres modules standards n'ont pas été utilisés soit parce qu'ils ne sont pas nécessaires pour la mission, soit parce que leur mode de fonctionnement est incompatible avec une exécution en parallèle. Nous avons aussi développé et ajouté trois autres modules:

- **`EXPLOR`**, qui calcule la position où il y a une plus forte probabilité de trouver des nouveaux objets et le positionnement idéal du robot pour saisir un objet détecté;
- **`ARM`**, qui simule le contrôle d'un bras manipulateur (fonctionnalité à être très rapidement ajoutée à *Hilare* 2); et
- **`VISUAL`**, qui simule la modélisation de l'environnement et la détection d'objets à partir d'images vidéo.

Dans ces nouveaux modules, comme l'objectif de l'expérimentation est d'étudier le fonctionnement du système de contrôle et non les algorithmes des modules eux-mêmes, nous avons mis en place des procédures simples mais qui ont des caractéristiques temporelles et un jeu de requêtes et répliques intéressants pour exhiber certains comportements, notamment au niveau de l'exécutif.

5.2.1 La locomotion

Les déplacements du robot ont été effectués par le module `PILO` [Fleury 96], en utilisant les requêtes présentées dans le tableau 5.1. Ce module, à partir d'un chemin géométrique de référence, calcule et exécute une trajectoire qui respecte les contraintes cinématiques (non holonomic) et dynamiques (limitations en vitesse et en accélération) du véhicule.

TAB. 5.1 – *Les requêtes du module* `PILO`

Requête	Type	Incompatibilités	Action
`avoidOnOff`	Contrôle		Active/désactive l'évitement local d'obstacles
`setDynamic`	Contrôle		Fixe les paramètres dynamiques (vitesse et accélération maximales) par défaut
`execPosterTraj`	Exécution	`execPosterTraj`	Exécute une trajectoire lue dans un poster

Au niveau du module `PILO`, l'exécution de la trajectoire consiste à produire et exporter dans un poster une consigne qui court le long de la trajectoire et sur laquelle devra s'asservir le robot. Le suivi de la consigne, quand le mode d'évitement local d'obstacle est activé,

met en opération (par des requêtes envoyées directement aux modules correspondants) d'autres fonctions dans AVOID, US et LOCO, avec les flux de contrôle (les requêtes envoyées entre les modules) et de données (les valeurs exportés dans les posters) indiqués dans la figure 5.3.

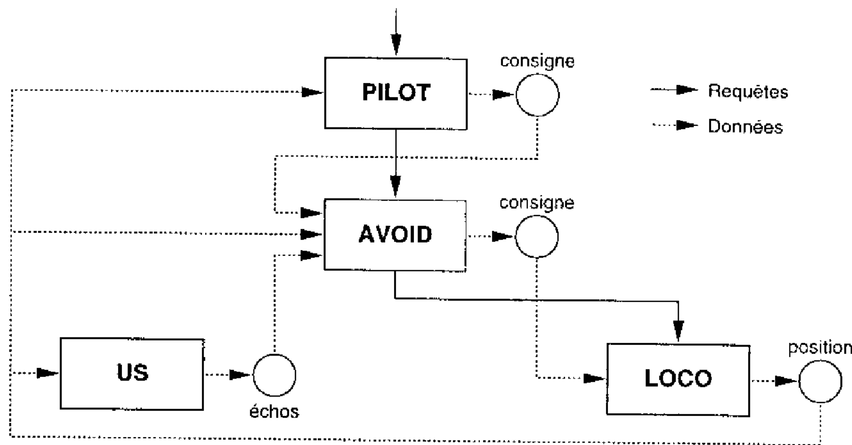


FIG. 5.3 – *L'organisation mise en place pour l'exécution d'une trajectoire*

Le module PILO permet d'exécuter quatre types de trajectoires, dont le tracé initial et les écarts tolérés sont transmis en paramètre de la requête d'exécution correspondante: des segments de droite, des arcs de cercle, des lignes brisées et des chemins de Reeds&Shepp (composés de segments de droite et d'arcs de cercle, et qui sont utilisés dans divers planificateurs de trajectoire). 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. Dans tous les cas, la trajectoire dynamique à exécuter est déterminée [Fleury 95] 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 de référence au delà d'un seuil fixé; et
- respecter les sens de translation et de rotation imposés.

Le module AVOID permet au robot d'évoluer sans s'arrêter dès qu'un obstacle se présente. L'évitement dynamique des obstacles se fonde sur le principe des champs de potentiel [Khatib 86, Khatib 95]: la consigne crée une force attractive et les obstacles des forces répulsives en fonction de la distance au robot. Les obstacles sont détectés au moyen d'ultrasons. La consigne filtrée est exportée dans le poster d'AVOID, où, en raison d'une requête locoTrack (page 38) émise par AVOID, elle sera lue par le module LOCO et utilisée pour l'asservissement en position. La loi de commande intégrée dans LOCO prend en compte les particularités de la cinématique d'*Hilare* 2 [Fleury 96].

5.2.2 La planification de trajectoires

Le modèle cinématique d'*Hilare* 2 fait apparaître des contraintes non holonomes (restrictions exprimées par des équations non intégrables qui incluent les dérivées des paramètres de l'espace de configurations). Cela implique qu'un chemin sans collisions ne correspond pas nécessairement à une trajectoire exécutable par le robot. Un planificateur de trajectoire pour ce genre de véhicule doit donc prendre en compte non seulement les restrictions géométriques (obstacles) de l'environnement mais aussi les contraintes cinématiques du robot.

Le module PLAN d'*Hilare* 2[1] utilise le planificateur de trajectoire pour robots mobiles non holonomes décrit dans [Laumond 94]. L'algorithme est basé sur la subdivision récursive d'un chemin sans collisions, généré par un planificateur géométrique de plus bas niveau qui ignore les contraintes cinématiques. Les buts intermédiaires le long de ce chemin initial sont choisis de sorte à qu'ils puissent être reliés entre eux par des chemins de Reeds&Shepp élémentaires (qu'on sait de longueur minimale) sans heurter d'obstacles. La trajectoire obtenue (une séquence de chemins de Reeds&Shepp) est ensuite optimisée pour réduire le nombre de manœuvres et la longueur totale.

Le planificateur géométrique de PLAN utilise un modèle de l'environnement où les obstacles sont représentés par des polygones. Ce modèle peut être dynamiquement généré par un autre module (et lu dans un poster) ou statique (décrit dans un fichier), option retenue dans cette expérimentation. Le tableau 5.2 présente l'ensemble des requêtes de PLAN nécessaires à la mission: la requête de replanification, quand l'environnement ne change pas entre deux planifications, permet de supprimer la phase initiale du traitement.

TAB. 5.2 – *Les requêtes du module PLAN*

Requête	Type	Incompatibilités	Action
autoBox	Contrôle		Choix de la détermination automatique ou manuelle des frontières de l'environnement
setPlannerEnv	Contrôle		Fixe les frontières de l'environnement
readModelFile	Exécution	plan, replan readModelFile	Lit le modèle (fichier) des obstacles dans l'environnement
plan	Exécution	plan, replan readModelFile	Planifie une trajectoire entre deux positions données
replan	Exécution	plan, replan readModelFile	Planifie une trajectoire entre deux positions données (utilise le même modèle de la planification précédente)

5.2.3 Le contrôle du bras manipulateur

Le contrôle intelligent d'un bras manipulateur monté sur le robot doit fait intervenir un ensemble de modules, avec au moins deux d'entre eux (similaires à LOCO et PILO) qui se chargent de l'asservissement de l'angle des articulations et de la génération de

1. La mise sous format de module de l'algorithme de planification a été faite par Maher Khatib.

trajectoires. Pour une opération plus performante, d'autres modules peuvent se charger de la planification de trajectoires sans collision, du suivi d'objets, etc.

Dans cette expérimentation, où les méthodes effectivement utilisées pour contrôler le bras ne sont pas fondamentales, on ne simule que le comportement d'un seul module, ARM, un hypothétique client des autres modules qui agissent sur le bras. Comme on l'indique dans le tableau 5.3, il possède des requêtes pour conduire le bras jusqu'à l'objet, le faire revenir à sa position de repos et contrôler la pince de saisie. Une approche possible pour la préhension d'objets, sur laquelle des travaux sont en cours au LAAS, est de combiner une estimation initiale de la position de l'objet avec un asservissent en temps réel, à partir d'images vidéo produites par une caméra placée à l'extrémité du bras.

TAB. 5.3 – *Les requêtes du module ARM*

Requête	Type	Incompatibilités	Action
`goObject`	Exécution	`goObject, goBack`	Conduit le bras à l'objet
`goBack`	Exécution	`goObject, goBack`	Conduit le bras à la position de repos
`gripState`	Contrôle		État de la pince (ouvrir et fermer)

5.2.4 Perception et modélisation de l'environnement

Le module VISUAL, qui possède les requêtes indiquées dans le tableau 5.4, simule la perception et la modélisation de l'environnement faite par une caméra vidéo.

TAB. 5.4 – *Les requêtes du module VISUAL*

Requête	Type	Incompatibilités	Action
`initModel`	Contrôle	`readObstacles` `searchObject`	Initialise le modèle (fiabilité des cellules et position des objets détectés)
`readObstacles`	Exécution	`readObstacles` `searchObject`	Lit le modèle (fichier) des obstacles dans l'environnement
`moveCamera`	Contrôle		Réoriente la caméra
`takeImage`	Contrôle	`searchObject`	Acquiert une image et la stocke
`searchObject`	Exécution	`searchObject`	Analyse une image; retourne le nombre d'objets trouvés et stocke leurs positions.
`getObject`	Contrôle		Retourne l'objet détecté le plus proche

Le modèle qu'il génère est basé sur une division de l'environnement en un nombre fixe de cellules élémentaires où, pour chaque cellule (i, j), les informations suivantes sont déterminées à partir des images acquises:

- Si la cellule est libre ou dans un obstacle.
- La certitude $c_{i,j}$ (entre 0 et 1) sur le fait que dans cette cellule il y a ou il n'y a pas un objet du type recherché (initialement 0 pour toutes les cellules).

La modélisation des obstacles est statique et faite à partir de la lecture du même fichier de description de l'environnement utilisé par le module de planification. La requête

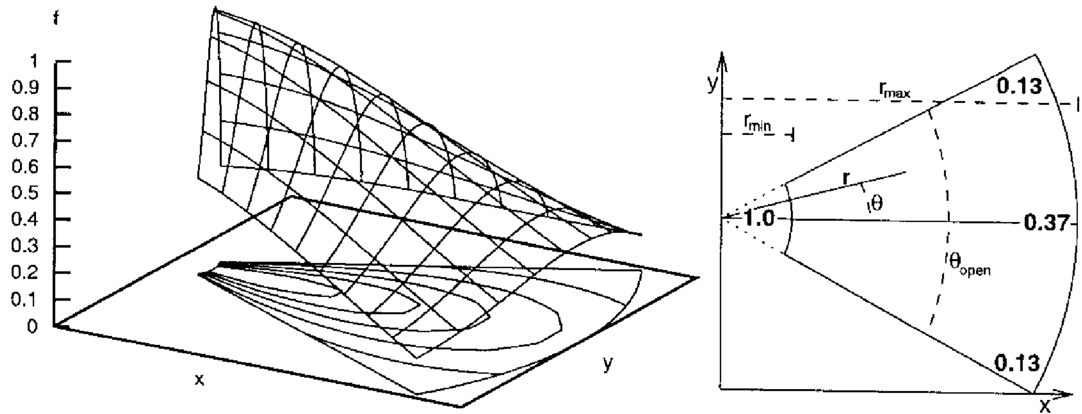


FIG. 5.4 – *Capacité de détection d'un objet selon sa position dans l'image*

`readObstacles`, à être appelée au début de la mission, calcule les cellules qui ont une intersection avec les obstacles polygonaux et les marque comme non libres[2]: cette information sera gardée tout au long de l'expérimentation.

À l'initialisation, les objets sont placés de manière aléatoire. La simulation de l'analyse des images suppose que la capacité de détection des objets diminue de façon exponentielle avec la distance et l'angle d'écart par rapport à l'axe central de la caméra (fig. 5.4). Si θ_{open} est l'angle d'ouverture du champ de vision de la caméra et $r_{\min}$ et $r_{\max}$ sont les distances minimale et maximale auxquelles un objet peut être perçu, la fiabilité f de la réponse de l'algorithme de détection quant à l'existence ou non d'un objet à la position (r, θ) est de

$$f(r,\theta) = \begin{cases} e^{-\left(\frac{r}{r_{\max}}\right)^2 - \left(\frac{2\theta}{\theta_{\text{open}}}\right)^2} & : \quad r_{\min} \leq r \leq r_{\max} \quad \text{et} \quad -\theta_{\text{open}} \leq 2\theta \leq \theta_{\text{open}} \\ 0 & : \quad \text{le cas contraire} \end{cases}$$

Quand une image est analysée, la simulation détecte initialement quels objets (parmi ceux disposés au hasard dans l'environnement) sont visibles dans la surface couverte par cette image, prenant compte des occlusions créées par les obstacles. En suite, chacun de ces objets pourra être détecté ou non: il est détecté si un numéro entre 0 et 1, généré de manière aléatoire, est inférieur à la probabilité de détection qu'on associe à cet objet. Cette probabilité est donnée par la fiabilité f de l'algorithme de détection pour la position du centre de la cellule où l'objet se trouve.

Après une analyse, le module met à jour son modèle de l'environnement. Pour chaque cellule dans le champ de vision de la caméra, étant r et θ la position et la direction du centre de la cellule par rapport à la position de la caméra au moment où l'image a été prise, la nouvelle certitude $c_{i,j}(t+1)$ quant à la présence d'un objet dans la cellule est:

$$c_{i,j}(t+1) = c_{i,j}(t) + [1 - c_{i,j}(t)]\, f(r,\theta)$$

Ce calcul augmente la certitude pour les cellules visualisées dans l'image. Cette augmentation est proportionnelle au degré de fiabilité $f(r,\theta)$ de l'algorithme de détection.

2. Cette routine a été fournie par Maher Khatib.

5.2.5 L'exploration

À partir du modèle généré par VISUAL, le module EXPLOR, dont les requêtes sont présentées dans le tableau 5.5, doit déterminer:

- la meilleure position pour explorer l'environnement (cellule d'où les images acquises permettront d'obtenir la plus grande augmentation de la certitude du modèle); et
- le positionnement du robot pour saisir un objet dont on connaît la localisation.

TAB. 5.5 – *Les requêtes du module EXPLOR*

Requête	Type	Incompatibilités	Action
calcGoal	Exécution	calcGoal	Calcule une nouvelle position d'exploration.
calcObj	Exécution	calcObj	Calcule la position du robot pour saisir un objet

L'algorithme de détermination de la position d'exploration utilisé est le suivant:

1. Grossir les obstacles pour prendre en compte la taille du robot et pouvoir le réduire à un point. Ce grossissement peut être surdimensionné, étant donné que la probabilité que les cellules proches des obstacles soient la meilleure position d'observation est réduite, en raison de l'occlusion créée par l'obstacle.
2. Prendre chacune des cellules libres comme cellule candidate:
   (a) Dans un rayon r_{max} autour de la cellule candidate, évaluer chacune des cellules (i, j) non cachées par des obstacles :
      i. Calculer $f(r, 0)$, la meilleure fiabilité possible de détection d'objets dans la cellule évaluée à partir de la cellule candidate.
      ii. Ajouter $(1 - c_{i,j})\, f(r, 0)$ à l'augmentation du niveau de certitude de la cellule candidate.
3. Choisir la cellule candidate qui a généré la plus grande augmentation du niveau de certitude du modèle.

Cet algorithme tend à éloigner le robot de sa position courante et à le conduire aux régions non encore explorées, étant donné que l'acquisition d'images augmente la certitude des cellules avoisinantes. Il garantit que la cellule choisie est optimale, mais impose une charge de calcul assez élevée au système.

On peut réduire significativement la quantité de calcul si on ne choisit pas toutes les cellules comme candidates, mais seulement quelques unes, bien distribuées dans l'environnement (une possibilité serait de prendre une cellule à chaque $r_{max}/2$). Selon le même raisonnement, on peut ne pas évaluer toutes les cellules au tour de la cellule candidate, mais seulement celles sur un nombre fixe d'axes radiaux (un à chaque $\theta_{open}/2$, par exemple).

La détermination de la position du robot pour qu'il puisse saisir un objet dépend des caractéristiques de l'objet, de l'emplacement du bras sur le robot et de l'algorithme qui conduit le bras jusqu'au objet. Dans cette expérimentation, nous supposons que les objets ne sont pas d'obstacles pour le mouvement du robot (ils sont par exemple placés plus haut que le véhicule) et que le centre du robot doit être dans le centre de la cellule de l'objet à saisir. La requête retourne une erreur si la cellule en question n'est pas atteignable.

5.3 L'exécutif

Pour cette mission, l'exécutif doit fournir au niveau tâche les services indiqués dans le tableau 5.6. Les sept premiers sont seulement des initialisations, appelées une fois pour toutes au début de l'exécution de l'expérimentation. Les autres correspondent aux fonctionnalités nécessaires à la mise en œuvre de l'algorithme de la figure 5.1 page 102.

TAB. 5.6 – *Les services offerts par l'exécutif pour l'expérimentation*

Service	Fonction	Incompatible avec[3]	Action
AVOID-ON	piloAvoidOnOff		
SET-DYNAMIC	piloSetDynamic		
AUTOBOX-OFF	planAutoBox		
SET-BOX	planSetPlannerEnv		
READ-MODEL[3]	planReadModelFile	CALC-TRAJ-GOAL CALC-TRAJ-OBJ	Interrompt
INIT-MODEL	visualInitModel	READ-OBST SEARCH-OBJ	Interrompt
READ-OBST[3]	visualReadObstacle	SEARCH-OBJ	Interrompt
EXEC-TRAJ-GOAL[3]	piloExecPosterTraj	CALC-TRAJ-OBJ EXEC-TRAJ-OBJ CALC-OBJ ARM-OBJ	Attend fin
EXEC-TRAJ-OBJ[3]	piloExecPosterTraj	EXEC-TRAJ-GOAL	Interrompt
CALC-TRAJ-GOAL[3]	planReplane	CALC-TRAJ-OBJ READ-MODEL	Attend fin
CALC-TRAJ-OBJ[3]	planReplane	CALC-TRAJ-GOAL EXEC-TRAJ-GOAL	Interrompt
		READ-MODEL	Attend fin
ARM-OBJ[3]	armGoObject	ARM-BACK EXEC-TRAJ-GOAL	Interrompt
ARM-BACK[3]	armGoBack	ARM-OBJ	Attend fin
OPEN-GRIP	armGripState		
CLOSE-GRIP	armGripState		
TURN-CAMERA	visualMoveCamera		
TAKE-IMAGE	visualTakeImage	SEARCH-OBJ	Attend fin
SEARCH-OBJ[3]	visualSearchObject	READ-OBST	Attend fin
GET-NEAR-OBJ	visualGetObject		
CALC-GOAL[3]	explorCalcGoal	EXEC-TRAJ-OBJ	Attend fin
CALC-OBJ[3]	explorCalcObj	EXEC-TRAJ-GOAL	Interrompt

Pour ce qui concerne les entrées et sorties des services, on peut mentionner les deux cas le plus intéressants:

1. **TURN-CAMERA** utilise la requête **visualMoveCamera** pour faire tourner la caméra, en adoptant comme paramètre d'entrée, à chaque appel, une orientation angulaire espacée d'une valeur constante par rapport à l'orientation précédente.
2. SEARCH-OBJ transmet au niveau décisionnel le nombre d'objets détectés retourné par la requête **visualSearchObject**, donnée nécessaire pour la prise de décisions.

3. Tous les services basés sur des requêtes d'exécution sont aussi incompatibles avec eux mêmes, et l'action correspondante est d'interrompre l'instance précédente.

Quant aux autres services, les résultats d'exécution sont stockés dans des variables et/ou dans des posters internes au niveau fonctionnel, où ils peuvent être lus et utilisés comme paramètres d'entrée par les autres services qui en ont besoin.

Le graphe d'incompatibilités de la figure 5.5), où les lignes brisées indiquent des attentes et les lignes continues, des interruptions, montre que six groupes de services (quelques-uns avec un seul service) sont nécessaires pour représenter les situations de conflit. Le tableau 5.7 liste les états possibles pour chacun des groupes.

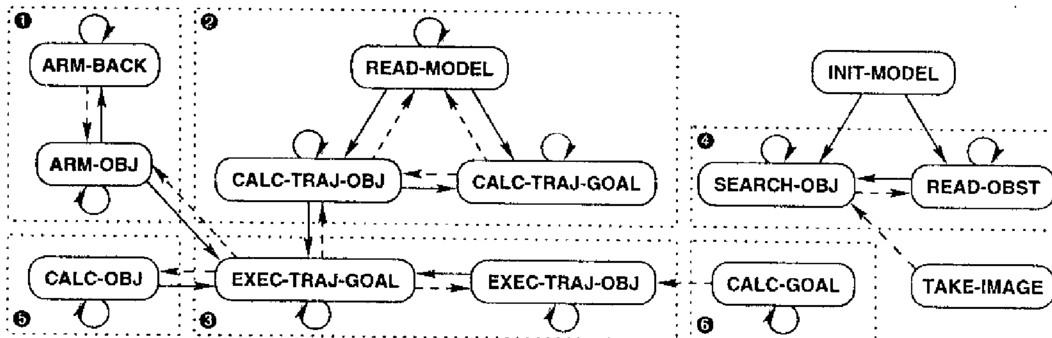


FIG. 5.5 – *Le graphe d'incompatibilités entre les services*

Groupe de services	États possibles
1. armGroup	ARM-BACK ARM-OBJ IDLE
2. planGroup	READ-MODEL CALC-TRAJ-OBJ CALC-TRAJ-GOAL IDLE
3. piloGroup	EXEC-TRAJ-OBJ EXEC-TRAJ-GOAL IDLE
4. visualGroup	READ-OBST SEARCH-OBJ IDLE
5. objGroup	CALC-OBJ IDLE
6. goalGroup	CALC-GOAL IDLE

TAB. 5.7 – *Les groupes de services*

La mise en œuvre de l'exécutif, dans le système d'exploitation en temps réel VxWorks, a été faite avec trois tâches en parallèle qui communiquent par mémoire commune:

- Réception des requêtes
- Envoi des répliques
- Boucle principale

Les deux premières font la communication avec le niveau tâche, alors que la troisième exécute les cycles de l'exécutif proprement dit. La raison de cette séparation est que, comme la communication avec PRS est faite par envoi de messages, son temps d'exécution est non borné et pourrait ralentir le cycle principal de l'exécutif.

La tâche qui reçoit les requêtes fait la vérification de syntaxe et, le cas échéant, exécute l'éventuelle fonction d'entrée associée au service. Ces fonctions servent essentiellement à faire des lectures de données exportées et des conversions mineures de format: en aucun

cas elles ne doivent exécuter des algorithmes complexes, envoyer des répliques, etc. Une fois que les paramètres d'entrée sont bien formés, l'activité est mise dans la liste d'activités et ensuite la tâche de lecture des requêtes réveille (par libération de sémaphore) la boucle principale de l'exécutif, qui prend en compte la nouvelle requête.

Pour les répliques, une stratégie de synchronisation du type producteur/consommateur *bufferisée* est mise en place entre la boucle principale de l'exécutif et la tâche qui envoie les répliques vers le niveau décisionnel. En cas de réplique finale indiquant une exécution réussie, la tâche d'envoi de répliques exécute ensuite une éventuelle fonction de sortie associée au service, qui prépare le résultat d'exécution à être envoyé au client du service.

La boucle principale fait le suivi de l'état d'exécution des activités et met en œuvre la politique de résolution de conflits. Son code C est fondamentalement généré à partir de la compilation de règles Kheops, développées selon les principes de la section 3.8. On en fournit deux exemples de règles dans la figure 5.6.

```
CalcTrajObjWhenExecTrajGoal:
  event == EventStart,
  service == "CALC-TRAJ-OBJ",
  @ExecTrajState == GOAL
==>
  do {changeInterState("EXEC-TRAJ-GOAL")},
  do {sendAbortRequest("EXEC-TRAJ-GOAL")},
  restrict status == StatusOK;
```

```
ArmGoWhenExecTrajGoal:
  event == EventStart,
  service == "ARM-OBJ",
  @ExecTrajState == GOAL
==>
  do {changeInterState("EXEC-TRAJ-GOAL")},
  do {sendAbortRequest("EXEC-TRAJ-GOAL")},
  restrict status == StatusWait;
```

FIG. 5.6 - *Exemple de règles de l'exécutif*

La première règle indique que le calcul de la trajectoire vers un objet détecté doit interrompre l'exécution d'une trajectoire d'exploration: pour cela, comme les deux fonctions appartiennent à des modules différents et ne sont donc pas intrinsèquement incompatibles, l'exécutif envoie une requête explicite d'interruption. Mais comme cette incompatibilité existe par des raisons d'efficacité, et non à cause d'une impossibilité réelle d'exécution simultanée, le démarrage de la deuxième activité est autorisé tout de suite (faisant `status == StatusOK`).

La situation est différente quand on veut saisir un objet avec le bras, où l'exigence d'immobilité est stricte. Dans ce cas, représenté dans la deuxième règle Kheops, la nouvelle requête doit attendre l'interruption effective du mouvement.

5.4 Le niveau tâche

C-PRS possède deux interfaces de programmation et de visualisation: une textuelle et l'autre graphique (un peu plus lente). Comme l'évolution temporelle du niveau tâche n'est pas très rapide, on peut ne pas l'embarquer sur le robot et le faire tourner sur une machine Unix distante, avec envoi de messages par réseau. Cela nous permet d'avoir une interface plus conviviale.

Les trois cycles de la figure 5.1 page 102 vont correspondre à trois procédures PRS, montrées dans la figure 5.7 page suivante. Dans la tâche de prise d'objets, le nombre d'objets déjà pris est gardé dans la variable `@TAKEN`. Dans la tâche de modélisation, après

l'acquisition de l'image, on lance en parallèle l'activité de recherche d'objets et la réorientation de la caméra, étant donné qu'elles ne sont pas incompatibles entre elles.

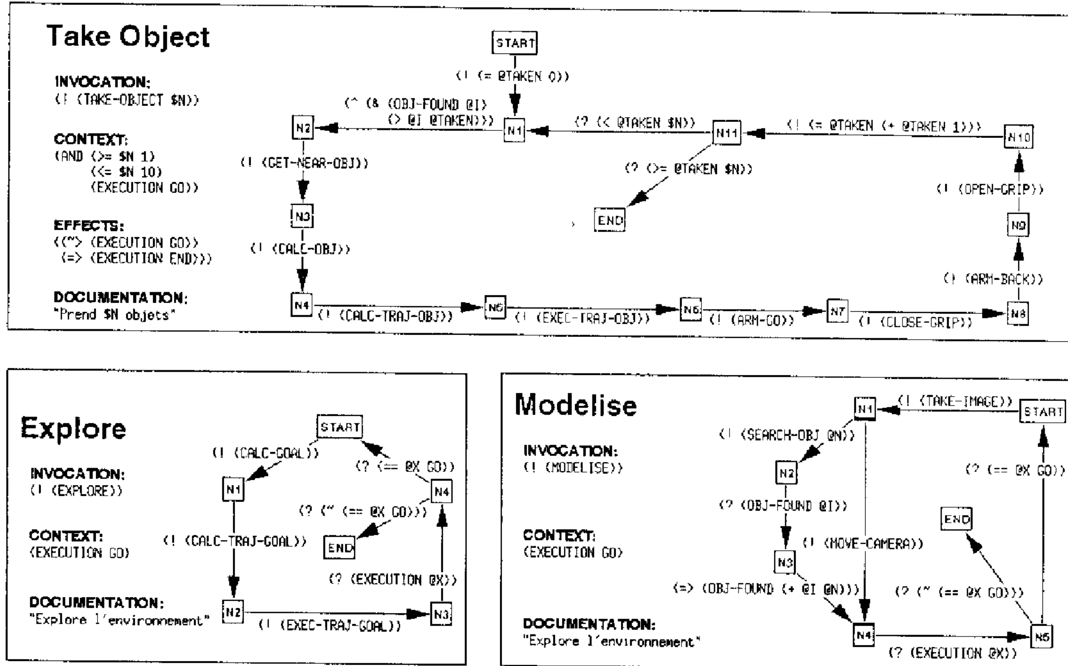


FIG. 5.7 - *Les procédures* PRS *pour les tâches de prise d'objet, exploration et modélisation*

Deux faits dans la base de données dont utilisés pour établir la communication entre les procédures:

- Le prédicat fermé `(EXECUTION $STATE)` - en faisant `$STATE` passer de `GO` à `END`, la tâche de prise d'objets signale que l'expérimentation est finie (le nombre désiré d'objets à été trouvé) et que les autres procédures peuvent finir leur exécution.
- Le fait fonctionnel `(OBJ-FOUND $N)` - le nombre d'objets déjà localisés, mis à jour par le cycle de modélisation.

Une procédure de base appelle les services d'initialisation des modules, établit le fait `(EXECUTION GO)` et, ensuite, lance en parallèle les buts (`TAKE-OBJECT`, `EXPLORE` et `MODELISE`) qui déclenchent les trois tâches principales. D'autres petites procédures (comme celles des figures 4.3 page 77 et 4.4 page 78) permettent d'accéder aux divers services de la couche fonctionnelle.

5.4.1 Vérification

Pour cet ensemble de services, les incompatibilités entre les services peuvent être intégrées dans le réseau de Pétri. On modélise les groupes de services par des places dont la couleur du jeton indique l'état du groupe. Comme l'aspect temporel n'existe pas dans

les réseaux de Pétri colorés classiques, on ne peut pas représenter le fait qu'un service interrompt un autre, mais seulement qu'ils ne peuvent pas s'exécuter simultanément. La figure 5.8 montre la représentation des incompatibilités pour le service EXEC-TRAJ-GOAL, qui ne peut s'exécuter que si le groupe execTrajGroup est IDLE, armGroup n'est pas go et calcTrajGroup n'est pas obj.

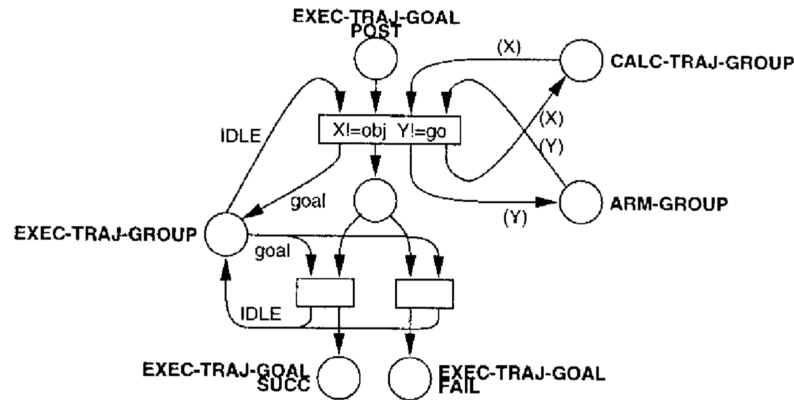


FIG. 5.8 – *Représentation des incompatibilités entre services*

La seule restriction pour que l'ensemble de procédures du niveau tâche puisse être modélisée par des réseaux de Pétri colorés est que le paramètre de retour du service SEARCH-OBJ (le nombre d'objets détectés après l'analyse d'une image) soit borné, ce qui normalement se vérifie dans des situations réelles. La figure 5.9 page suivante présente la partie du réseau de Pétri qui correspond à la KA de la tâche de modélisation.

L'analyse du réseau de Pétri peut être faite avec des outils automatiques, comme Design/CPN [CPN Group 96]. Pour cet exemple, l'analyse montre qu'il n'y a pas de *deadlock*: sauf en cas d'erreur d'exécution, l'exécution ne s'arrête qu'après que le nombre voulu d'objets ait été saisi. Elle permet de se rendre compte aussi de quelques points qui, dans une application importante, devraient être considérés:

- Il y a une possibilité de cycle éternel (qui est d'ailleurs évidente: il suffit que la perception ne trouve jamais aucun objet). Cette éventualité peut être éliminée par l'inclusion d'un compteur de nombre de cycles ou d'un détecteur de terminaison de la tâche après exploration complète.
- En cas d'erreur dans la procédure de prise d'objets, les autres ne s'arrêtent pas (le fait (EXECUTION GO) n'est jamais établi) et tombent en cycle éternel.
- Il y a la possibilité de commencer un cycle d'exploration alors qu'un objet attend pour être pris, en raison du fait que le dépôt de l'objet sur le robot n'est pas incompatible avec un mouvement. On illustre cette possibilité dans le scénario de la section 5.5.
- Si l'exigence que le robot s'arrête dès qu'il détecte un objet est très stricte, il faut prévoir un mécanisme pour éliminer la possibilité (certes petite et pratiquement sans conséquences) qu'une exécution de trajectoire d'exploration en attente démarre pendant les quelques instants entre la fin du calcul de la trajectoire vers l'objet et le

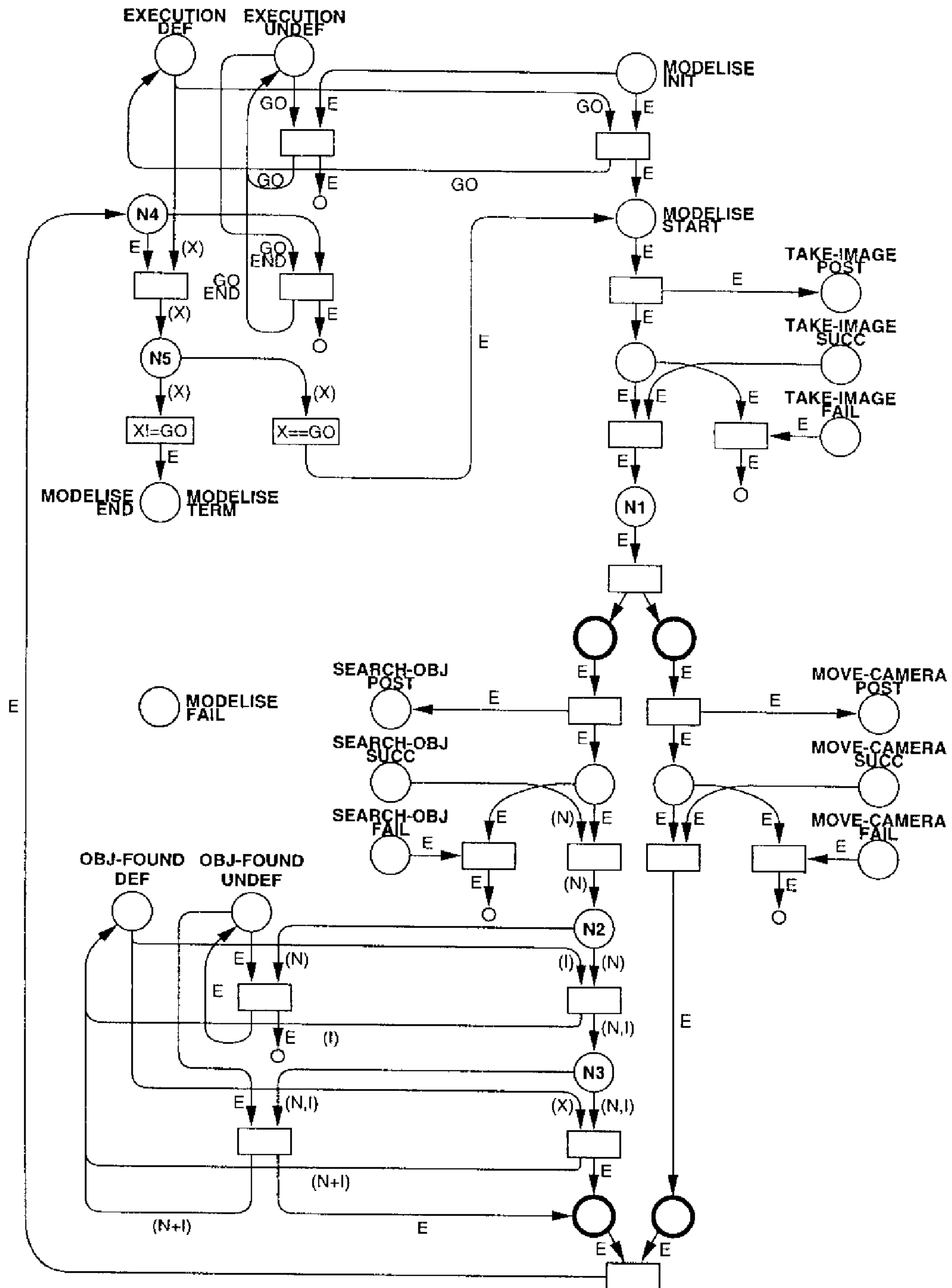


FIG. 5.9 – *Réseau de Pétri correspondante à la tâche de modélisation*

démarrage de l'exécution de cette trajectoire.

Tous ces petits problèmes semblent évidents **après** les avoir listés. Un des grands avantages d'utiliser une technique de vérification formelle est exactement la génération exhaustive des situations possibles, ce qui permet de faire apparaître des configurations auxquelles on n'avait pas pensé au moment de la conception du système.

5.5 Un scénario illustratif

Pour donner des exemples plus concrets sur l'exécution du système, la figure 5.10 page ci-contre représente quelques étapes d'un scénario d'exécution de l'expérimentation. Le robot doit collecter trois objets (sur un total de cinq existants, indiqués par les lettres de `A` à `E` sur le dessin). Les lignes brisées sont des trajectoires planifiées, alors que les lignes pleines indiquent des parcours exécutés. On indique aussi la taille du champ de vision en dehors duquel on considère que la fiabilité de la détection d'objets est nulle.

Au départ, comme peu de cellules ont été analysées, une position d'exploration proche du centre de l'aire totale est choisie (`1`). Pendant l'exécution de cette trajectoire (où à l'instant `t1` intervient un évitement local d'obstacles), l'objet `A` est localisé (`t2`). Le déplacement est interrompu et le système calcule une trajectoire vers la position de `A`. On liste la séquence d'événements qui se produisent autour de l'instant `t2`.

1. Le service `SEARCH-OBJ`, dans la procédure de modélisation, retourne `1`, indiquant qu'un objet a été détecté dans la dernière image analysée. Le fait `(OBJ-FOUND 1)` est conclu dans la base de données.
2. La tâche de prise d'objets, qui attendait qu'un objet soit détecté `(^(&(OBJ-FOUND @I) (>@I TAKEN)))`, poursuit son exécution: elle demande que l'objet détecté soit récupéré (`GET-NEAR-OBJ`) et sollicite le calcul d'une trajectoire jusqu'à la cellule de l'objet (`CALC-TRAJ-OBJ`).
3. En raison de la première règle de la figure 5.6 page 112, quand la requête de calcul de trajectoire arrive à l'exécutif, une demande d'interruption de la trajectoire d'exploration est envoyée au module.

La trajectoire calculée est ensuite exécutée. Après l'arrivée à la cellule de `A`, le bras est actionné pour prendre l'objet. La caméra continue à tourner pendant le mouvement du bras et ainsi `B` est détecté. Mais la procédure de prise d'objet ne prend pas tout de suite en compte cet événement, étant donné qu'un nouveau test de l'existence d'objets détectés ne sera fait qu'après la conclusion du cycle courant.

Ainsi, après la saisie de `A` et pendant son dépôt sur le robot (qui n'est pas incompatible avec un déplacement exploratoire), la tâche d'exploration peut redémarrer: elle calcule un nouveau point d'observation (`2`), une trajectoire pour y aller et commence à l'exécuter.

À la fin du dépôt de l'objet (`t3`), une nouvelle attente d'un objet détecté se met en place. Comme il y en a un déjà disponible (`B`), la tâche de prise d'objets se poursuit, avec l'interruption de l'exploration. La trajectoire vers `B` est calculée et exécutée.

Après la saisie de `B`, la tâche d'exploration redémarre. Le nouvel objectif (`3`) est déterminé et la trajectoire qui y conduit exécutée après calcul. Comme aucun nouvel objet n'a

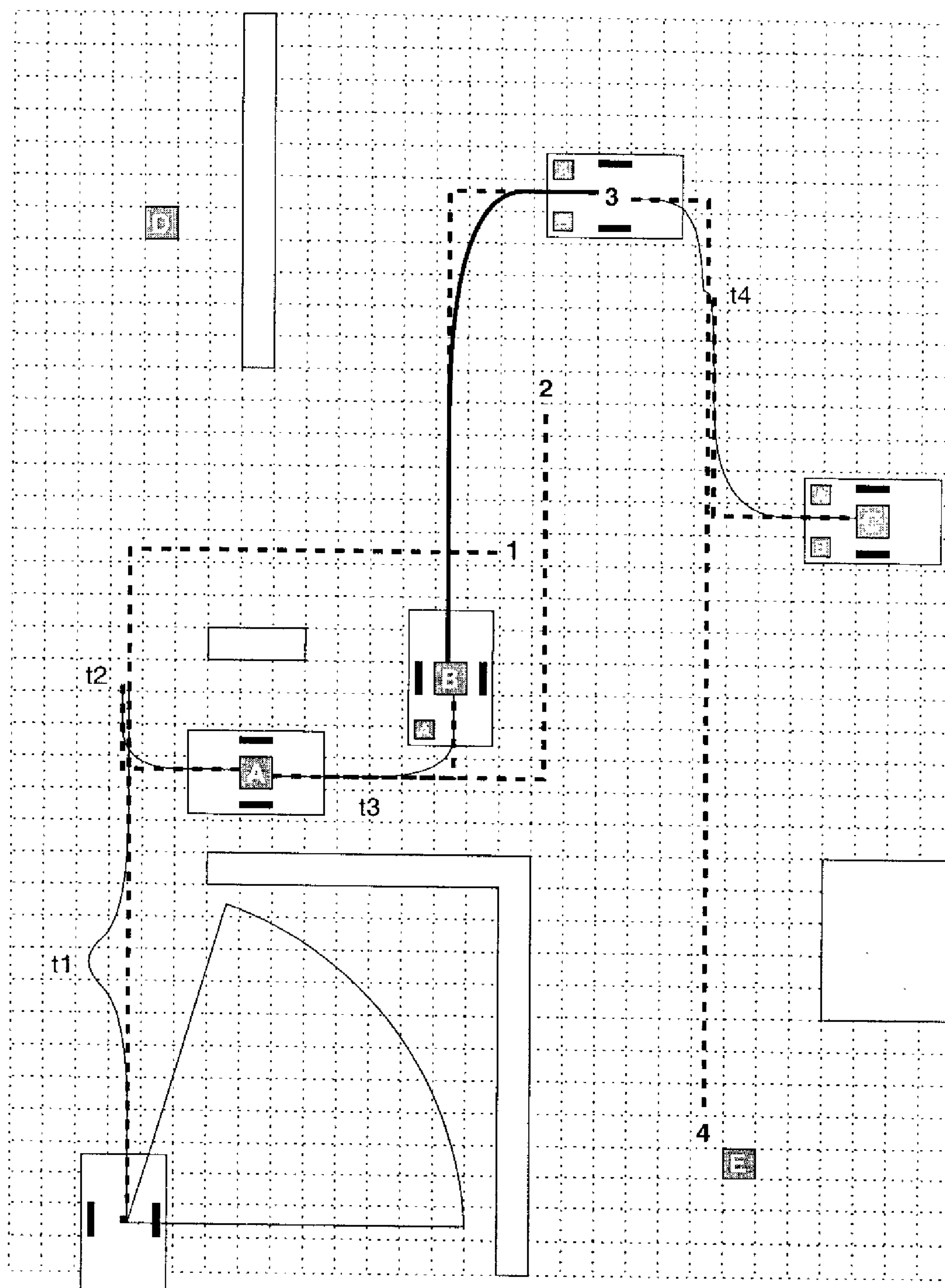


FIG. 5.10 – *Un scénario d'exécution de l'expérimentation*

été découvert, une nouvelle position d'observation (4) est choisie. Pendant ce mouvement, un nouvel objet, C, est localisé (t4), ce qui, après calcul de trajectoire, déplacement et saisie, marque la fin de l'exécution des procédures.

5.6 Un exemple d'exécution

L'exécution de l'expérimentation peut être accompagnée à l'aide d'une interface graphique qui a été développée et qui permet de visualiser:

- la position du robot;
- les obstacles;
- la position des objets non détectés (possibilité qui n'existe qu'en simulation), détectés mais pas encore récupérés (avec `visualGetObject`) et récupérés;
- la cellule qui correspond à la position optimale d'observation;
- la dernière trajectoire planifiée; et
- le modèle de l'environnement. On indique avec des couleurs différentes les cellules qui ont une certitude:
  - inférieure à 1/3;
  - entre 1/3 et 2/3; et
  - supérieure à 2/3.

La figure 5.11 page ci-contre permet d'avoir un échantillon du type d'information fourni par cette interface. Dans cette expérimentation, réalisée dans une grande salle au LAAS, l'environnement à explorer est divisé en cellules de 30×30cm. *Hilare* 2 a pu retrouver les cinq objets virtuels, sur un total de dix dispersés dans la salle, qu'on lui avait demandé de trouver. Les modules et l'exécutif tournent sur les cartes VxWorks embarquées sur *Hilare* 2, alors que PRS et l'interface s'exécutent sur une station de travail.

5.7 Conclusion

L'expérimentation a permis de montrer:

- l'adéquation du modèle d'exécutif et des protocoles de communication proposés aux conditions d'opération en temps réel;
- la possibilité de réaliser des missions d'un certain degré de complexité sans que le niveau décisionnel soit obligé de manipuler les données complexes; et
- l'utilité des procédures de vérification formelle non seulement pour apporter des preuves de certains comportements corrects, mais aussi pour aider à valider les algorithmes mis en place.

L'intégration des principes qui ont été proposés dans ce travail dans une des prochaines grandes expérimentations du groupe de recherche permettra de juger d'une manière plus précise son adéquation aux systèmes robotiques complexes.

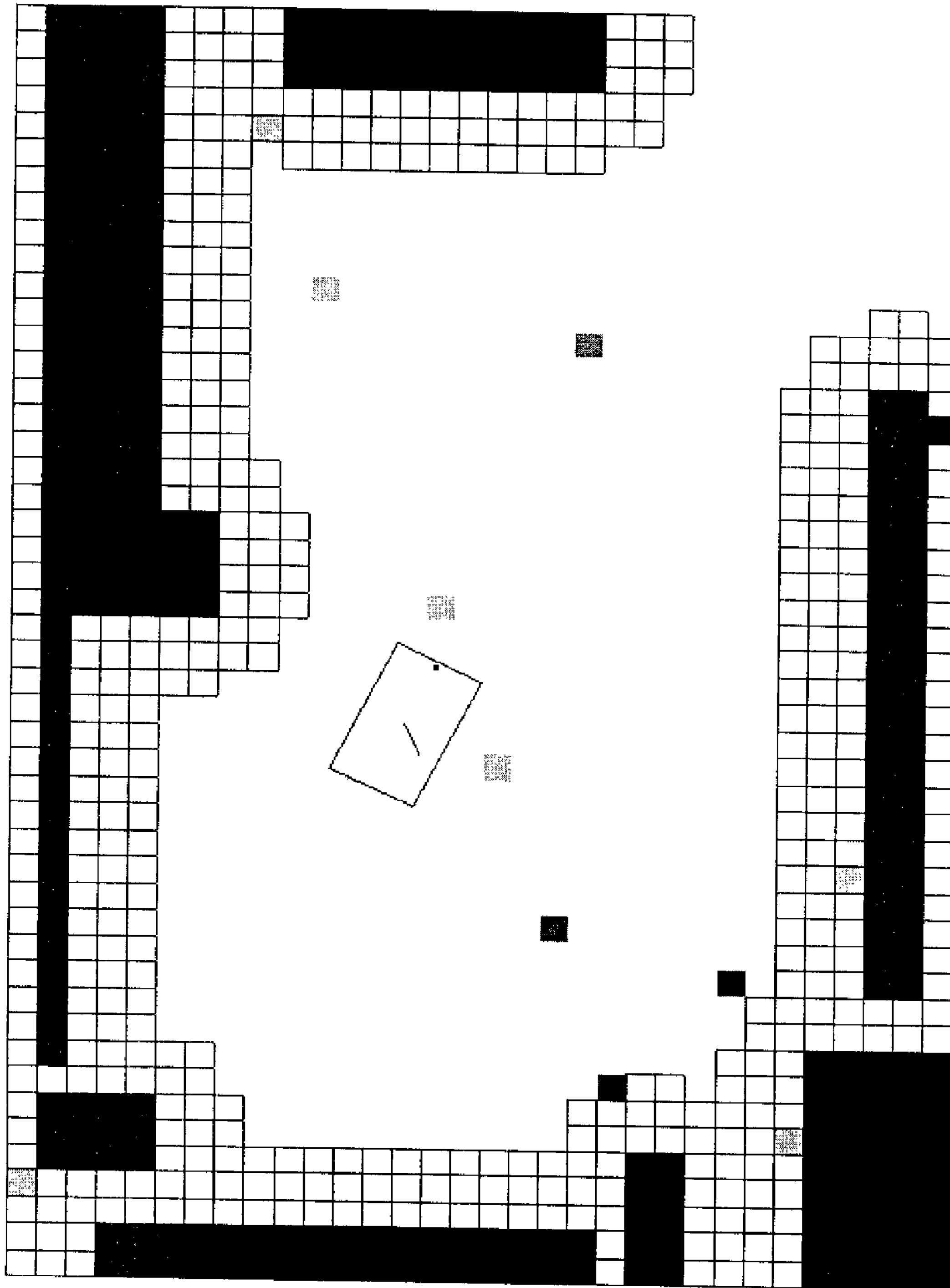


FIG. 5.11 - *Un état du système pendant une expérimentation*

Conclusions et perspectives

es contributions originales présentées dans ce travail se concentrent dans trois domaines principaux:

Meilleur définition de l'architecture LAAS – un des points importants dans les organisations hiérarchisées est la détermination des limites entre les divers éléments et entre les niveaux de la hiérarchie. Nous avons proposé des critères objectifs pour fixer plus précisément certaines frontières à l'intérieur de l'architecture LAAS:

- Le critère de l'utilisation prévisible et persistante des ressources (section 2.6) détermine le niveau de granularité des fonctions de modules. Le même critère indique les activités qui peuvent être déclenchées directement par le niveau fonctionnel et celles qui dépendent du niveau décisionnel.
- Le graphe d'incompatibilités (section 3.6.1) formalise l'idée intuitive qui existait déjà à propos de l'étendue appropriée pour un module (les fonctions d'un module doivent correspondre à un groupe de services).
- La séparation entre données dénombrables et non dénombrables délimite les informations traitées par les niveaux fonctionnel et décisionnel.
- Des protocoles de communication mieux définis rendent plus stable l'interface entre les modules et l'exécutif (section 2.5) et entre l'exécutif et le niveau tâche (section 3.9).

L'exécutif – Même si l'exécutif faisait déjà partie du concept de l'architecture LAAS, il n'existait pas jusqu'à maintenant comme un élément individualisé dans ses mises en œuvre. Nous proposons un modèle pour l'exécutif dans lequel le temps d'exécution est borné et certaines propriétés logiques sont démontrées, grâce à l'utilisation du système Kheops.

Vérification du niveau tâche – L'existence d'un exécutif qui se charge de la manipulation des données brutes rend envisageable une vérification logique du niveau tâche. Nous présentons quelques résultats concernant la vérification d'un ensemble de procédures PRS par les réseaux de Pétri colorés.

Plusieurs perspectives de continuation et d'approfondissement de ces travaux existent. On peut citer les plus prometteuses d'entre elles:

- Spécification plus détaillée et généralisée des divers protocoles de communication entre les éléments de l'architecture.

- Intégration entre la génération des modules, faite par G^{en}oM, et l'exécutif. La description formelle des modules contient les informations sur les incompatibilités entre fonctions d'un même module, ce qui permettrait de générer automatiquement une partie des règles Kheops de l'exécutif. La syntaxe de G^{en}oM peut aussi être étendue pour décrire les incompatibilités entre modules; dans ce cas, une bonne partie des règles et le graphe d'incompatibilités pourraient être générés automatiquement.
- Augmentation du pouvoir de vérification de Kheops. Dans sa mise en œuvre actuelle, Kheops ne vérifie que le résultat d'un seul parcours du graphe de décision. Dans des applications cycliques comme l'exécutif, et pour des variables d'entrée qui suivent une certaine trajectoire, des comportements à plus long terme peuvent être prouvés [Kaestner 92].
- Définition d'un plus grand nombre de modèles d'équivalence entre PRS et les réseaux de Pétri. Les modèles que nous avons déduits sont basés sur les effets extérieurs des actions de PRS. Si on modélise à un niveau plus bas le fonctionnement interne du système, d'autres fonctionnalités, comme les meta-KAs, deviennent représentables.
- Réalisation d'un convertisseur automatique entre un ensemble de procédures PRS et le réseau de Pétri équivalent.
- Utilisation des réseaux de Pétri temporisés pour la vérification du niveau tâche. Avec cet outil, outre la preuve des propriétés logiques, une certaine validation temporelle de l'ensemble de procédures PRS devient possible. Cette démarche présuppose qu'on peut modéliser, avec des bornes ou de façon stochastique, le temps d'exécution des diverses fonctions du niveau fonctionnel.

D'autres perspectives, principalement pour ce qui concerne l'incorporation de nouvelles capacités au sein de l'exécutif, seront mieux précisées par les expérimentations futures. La philosophie de développement de l'architecture LAAS a toujours privilégié les interactions entre les définitions théoriques et les résultats expérimentaux, ce qui devra être aussi le cas pour les évolutions futures de la stratégie de contrôle de l'exécution.

Annexe A

Le système Kheops

Comme tout système de production basé sur des règles, Kheops est formé par un ensemble de règles de production du type condition et action (la *base de règles*), une base de données dynamique (la *mémoire de travail*) et un interpréteur de règles (la *machine d'inférence*) qui évalue les règles de façon cyclique selon l'état de la mémoire de travail.

A.1 Les composants

A.1.1 La base de règles

Les règles de production Kheops peuvent être symboliquement représentées par des implications

$$C_1 \cdots C_l \to A_1 \cdots A_m$$

dans lesquelles $C_1 \cdots C_l$ représente un ensemble de conditions et $A_1 \cdots A_m$, un ensemble d'actions. Comme le déclenchement d'une règle consiste à réduire l'ensemble de valeurs possibles pour un ou plusieurs attributs, les actions Kheops sont appelées "réductions". Il y a d'autres actions possibles (certains appels à des fonctions externes), mais qui n'ont pas d'influence sur le processus de déduction.

A.1.2 La mémoire de travail

La mémoire de travail Kheops est composée d'un ensemble de faits qui décrivent la valeur d'un nombre fini d'attributs. Tous les attributs prennent leurs valeurs dans un ensemble de valeurs discrètes: les attributs symboliques le font dans un ensemble discret de constantes et les attributs numériques sont rendus discrets par partition de l'axe numérique en un ensemble d'intervalles exclusifs, de façon à pouvoir représenter tous les opérateurs numériques par l'appartenance à une union de ces intervalles exclusifs. Les attributs peuvent donc être représentés par des expressions du type $(a_s\ v_1 \cdots v_n)$ où:

- Si a_s est un attribut symbolique, $v_1 \cdots v_n$ sont des valeurs constantes dans l'ensemble de valeurs possibles pour l'attribut. Le fait $(a_s\ v_1 \cdots v_n)$ peut être interprété comme

$a_s \in \{v_1 \cdots v_n\}$.

- Si a_s est un attribut numérique, $v_1 \cdots v_n$ sont des intervalles exclusifs et l'assertion signifie $a_s \in v_1 \cup \cdots \cup v_n$.

À un moment donné du processus d'inférence, un attribut peut être bien connu (a_s v_i), mal défini (a_s $v_1 \cdots v_n$) ou inconnu. Le dernier cas, étant donné que le processus décisionnel est fermé, est équivalent à dire qu'il existe un fait mal défini (a_s $v_1 \cdots v_m$), où $\{v_1 \cdots v_m\}$ représente l'ensemble de toutes les valeurs possibles pour a_s.

A.1.3 La machine d'inférence

D'une façon générale, le moteur d'inférence dans un système de production exécute des itérations de détermination des règles conflictuelles, résolution du conflit et déclenchement de la règle sélectionnée. La machine d'inférence Kheops ne travaille que sur des données originalement présentes ou déduites dans la mémoire de travail ("*forward chaining*"): il n'y a ni génération ni décomposition de buts. Au long des itérations successives, la valeur des attributs est réduite de façon à arriver à la valeur la plus spécifique possible.

Kheops suppose un raisonnement monotone, ce qui signifie que des nouvelles déductions ne peuvent pas contredire des conclusions déjà acquises, mais seulement les raffiner. Cela implique que:

- si une règle devient valide dans une itération, elle demeure valide pour toutes les itérations suivantes.
- si une règle est déclenchée, elle peut être exclue de la base de règles, étant donné qu'un nouveau déclenchement n'apportera aucune nouvelle information. Cela implique que la machine d'inférence va s'arrêter après un nombre fini d'itérations, borné par le nombre de règles dans la base de règles.
- si on déclenche toutes les règles valides, leur ordre de déclenchement ne change pas le résultat final.

Quand la machine d'inférence Kheops exécute les actions d'une règle, elle réduit la valeur d'un ou plusieurs attributs dans la mémoire de travail, par intersection entre la restriction indiquée par la règle et la valeur courante de l'attribut. À titre d'exemple, prenons un attribut entier n dont l'univers de variation D_n a été partitionné de la façon suivante:

$$D_n = \{I_1, I_2, I_3\} \quad \text{où} \quad \begin{array}{rcl} I_1 & = &]-\infty \;\; 0 \;\; [\\ I_2 & = & [\;\; 0 \;\; 2 \;\; [\\ I_3 & = & [\;\; 2 \;\; +\infty [\end{array}$$

Si dans une itération le fait (n I_1 I_2) est présent dans la mémoire de travail et qu'on exécute une règle dont l'action est `(restrict n >= 0)`, la valeur de n sera réduite à I_2, faisant de lui un attribut bien défini.

A.2 La syntaxe

On présente un exemple de programme Kheops dans la figure 3.8 page 65. Une base de connaissance codée en Kheops consiste en:

1. un ensemble fini d'attributs toujours connus à l'entrée du processus de déduction (l'espace d'entrée de l'application);
2. un ensemble fini d'attributs (l'espace de sortie de l'application) dont les valeurs sont requises à la sortie du processus de déduction et qui doivent être déduits par enchaînement de règles à partir de l'espace d'entrée.
3. la base de règles;
4. facultativement, des déclarations de types et des initialisations d'attributs.

Les déclarations `inputs` et `outputs` déterminent les espaces d'entrée et de sortie de l'application. Les attributs qui ne font partie d'aucun de ces espaces sont dénommés attributs intermédiaires: ils existent soit pour faciliter l'écriture et/ou améliorer le style de programmation des règles (comme `diagn`), soit pour satisfaire aux contraintes de la logique propositionnelle (comme `difP`, nécessaire pour tester la valeur de `pres2 - pres1`).

Les règles de productions sont des `IF ... THEN ...` classiques, avec une syntaxe

identificateur: *condition* ==> *action*.

Les conditions peuvent être liées par des opérateurs logiques `OR` (`|`), `AND` (`,`) ou `NOT` (`!`). Les actions, qui peuvent être concaténées avec des `AND` (`,`), sont soit des restrictions faites par Kheops lui-même (`restrict`), soit des appels à des fonctions externes C (`do`) qui modifient ou non des attributs. Kheops prévoit aussi deux types spéciaux de règles:

Les règles de dépendance – indiquent au système les éventuelles combinaisons impossibles de valeurs pour les attributs d'entrée, ce qui permettra de réduire la taille du graphe de décision. Leur syntaxe est:
identificateur: `impossible` *condition*.

Les règles d'incomplétude – déclenchées quand il ne reste plus d'autres règles déclenchables mais il y a encore des attributs de sortie non définis. Leur syntaxe est:
identificateur: `else` *action*

Certaines facilités d'écriture ont aussi été prévues pour manipuler des structures, des vecteurs et des règles répétitives. Même si la syntaxe adoptée fait apparaître des entités qui ressemblent à des variables, la logique est toujours propositionnelle: ces "variables" doivent avoir des domaines finis explicites et seront éliminées par expansion des expressions où elles apparaissent pendant le pré-traitement de la base de connaissance. On donne quelques exemples dans le tableau A.1 page suivante.

TAB. A.1 – *Quelques exemples d'utilisation d'ensembles en* Kheops

EXEMPLE	ÉQUIVALENCE
`struct (level,alarm) sensor["1".."3"];`	
`inputs foreach i in ["1".."3"]` `(sensor[i].level);`	`inputs sensor-1-level, sensor-2-level,` `sensor-3-level;`

`r1: foreach color in ["red","pink","blue"]`
`(attrib != color)`
`==> ... ;`
	`r1: attrib != "red", attrib != "pink",`
	`attrib != "blue"`
	`==> ... ;`

`foreach i in ["1","3"]`
`( r[i]: sensor[i].level > 10.0`
`==> ... ;)`
	`r1: sensor-1-level > 10.0`
	`==> ... ;`
	`r3: sensor-3-level > 10.0`
	`==> ... ;`

Dans les cas où Kheops est utilisé de façon cyclique, les valeurs des attributs dans le cycle de déduction précédent peuvent être utilisées comme des entrées implicites dans le cycle de déduction courrant, en faisant référence à l'identificateur de l'attribut précédé d'un `@`. On doit nécessairement prévoir une valeur d'initialisation pour ces entrées implicites avec l'instruction `init`.

```
inputs zeroOrOne;    outputs out,stab;
init @out (false);
r1: zeroOrOne == 0
    ==> restrict !out;
r2: zeroOrOne != 0
    ==> restrict  out;
r3: (@out, out) | (!@out,!out)
    ==> restrict  stab;
r4: (@out,!out) | (!@out, out)
    ==> restrict !stab;
```

Kheops permet aussi d'exprimer certaines conditions liées à des instants ou à des intervalles de temps précédents. Le temps est traité de façon discrète, avec un nouveau instant additionné à chaque cycle de déduction. Le système maintient un historique de l'évolution des attributs, dimensionné de façon à ne garder que les informations nécessaires au raisonnement. Comme ces possibilités n'ont pas été utilisées dans nos travaux, on ne les présentera pas ici.

A.3 La compilation

Les deux étapes qui consomment la plupart du temps d'exécution de la majorité des systèmes de production à base de règles sont la détermination des règles applicables et le choix d'une d'entre elles à appliquer. Dans le contexte de Kheops, ces deux aspects ne sont pas critiques parce que:

- le langage est propositionnel: aucune reconnaissance de modèles ("*pattern matching*") n'est nécessaire pour accéder aux règles; et
- le raisonnement est monotone, ce qui fait que le choix de la règle à être déclenchée n'est pas important.

Ces deux restrictions sur la représentation de la connaissance (propositionnel) et sur le raisonnement (monotone) limitent le pouvoir d'expression du système, mais peuvent être utilisées pour "procéduraliser"entièrement l'interprétation des règles de production.

La première étape du pré-traitement de la base de règles est l'analyse des attributs. Par vérification exhaustive des constantes avec lesquelles les attributs sont comparées, le système détermine leur type (entier, symbolique, etc.) et leur domaine de variation (liste de constantes pour les symboliques, ensemble d'intervalles pour les numériques). Il déduit alors l'*espace observable* de l'application, composé de tous les attributs qui ont une influence sur la détermination des attributs de sortie:

1. les attributs explicitement déclarés comme attributs d'entrée;
2. les attributs dont la valeur dans le cycle précédent est utilisée dans les conditions de déclenchement des règles; et
3. les attributs intermédiaires dont la valeur devra être attribuée en temps réel par des appels à des fonctions C extérieures.

Le compilateur produit en suite une arborescence de décision dont les nœuds sont:

- des nœuds qui testent un attribut et qui ont autant de successeurs que les valeurs possibles pour cet attribut;
- des nœuds procéduraux où la valeur d'un attribut est déterminée par un appel à des fonctions externes (un seul successeur);
- des nœuds procéduraux qui appellent des fonctions externes mais sans conséquence sur les attributs (un seul successeur); ou
- des nœuds feuille (sans successeurs) avec les résultats de la décision.

À chaque nœud on associe une configuration de la mémoire de travail et l'ensemble de règles qui ont déjà été déclenchées au long du chemin de la racine jusqu'au nœud en question. Dans l'instance de la mémoire de travail associée au nœud, les attributs de l'espace observable sont classés en trois catégories:

- **testé** – attribut déjà testé (sur le chemin de la racine jusqu'au nœud en question il y a un nœud de test qui porte sur cet attribut).
- **testable** – attribut pas encore testé, mais qui peut l'être.
- **non testable** - ne peut pas encore être testé: attribut intermédiaire dont la valeur n'a pas encore été fixée (sur le chemin de la racine jusqu'au nœud en question il n'y a pas de nœud procédural qui fait appel à la fonction externe qui attribue sa valeur).

L'algorithme de génération d'une arborescence équivalente à l'ensemble des règles est basé sur les idées suivantes [Ghallab 88]:

1. Étant donné un nœud, avec la mémoire de travail associée, on peut déterminer l'ensemble des règles déclenchables qui n'ont pas encore été déclenchées au long du chemin de la racine jusqu'au nœud et simuler leur déclenchement. Cela conduit à:
    - la restriction de la valeur de certains attributs.
    - la création de nouveaux nœuds procéduraux, si les règles font appel à des fonctions externes. Si des attributs intermédiaires ont leur valeur attribuée par ces appels, ils passent de l'état non testable à testable.

2. S'il n'y a pas de règles déclenchables, on peut déterminer l'ensemble de règles valides. Ces règles ont des conditions qui soit sont satisfaites, soit font référence à des attributs testables. Pour les règles valides on a la certitude que, après un nombre fini de tests, elles vont devenir déclenchables. On doit alors choisir le "meilleur" attribut testable et construire un nœud de test, avec tous ses successeurs. On les étudie ensuite de façon récursive.
   Le choix de l'attribut testable est le point clef de la procédure: la stratégie utilisée va déterminer la profondeur moyenne des nœuds feuille. Comme il s'agit d'un problème d'optimisation NP-complet, Kheops adopte une méthode d'optimisation locale, basée sur des heuristiques.
3. La récursion s'arrête quand il n'y a plus de règles déclencheables ni de règles valides. À ce point on a atteint un nœud feuille de l'arborescence.

L'arborescence produite n'est pas optimale en taille parce qu'elle peut contenir des sous-arborescences identiques. Un algorithme d'optimisation la transforme ensuite en un graphe orienté sans circuit et avec une racine[1]. Dans la figure 3.8 page 65 on présente un exemple de graphe de décision: il est intéressant de noter que l'attribut intermédiaire `diagn` a complètement disparu du graphe.

L'interprétation des règles peut alors être réduite à une traversée d'un graphe, caractérisée par une borne supérieure du temps d'exécution déterminée par le chemin de la racine du graphe jusqu'à la feuille la plus profonde. Tous les attributs de l'espace observable sont testés au maximum une fois au long de n'importe quel chemin de la racine du graphe à un nœud feuille. La complexité de l'interprétation est limitée par le nombre de paramètres d'entrée, et est indépendante du nombre de règles.

L'enchaînement des règles doit mener à la détermination des valeurs de tous les attributs de l'espace de sortie et aucun déclenchement de règle ne doit causer de contradictions. Cependant, pendant la compilation les éventuelles configurations de l'espace d'entrée qui causent des inconsistances – un attribut réduit à un ensemble de valeurs possibles nul – et celles qui ne permettent pas la détermination de tous les attributs de sortie – des cas appelés d'incomplétude – sont identifiées.

1. Il ne s'agit pas d'une arborescence parce que le graphe peut ne pas être simple (avec des arcs en parallèle) et contenir des cycles.

Bibliographie

[Alami 93] Rachid Alami, Raja Chatila & Bernard Espiau. *Designing an Intelligent Control Architecture for Autonomous Robots.* In ICAR - International Conference on Advanced Robotics, pages 435–440, Tokyo, Japan, November 1993.

[Albus 89] James S. Albus, H. G. McCain & R. Lumia. *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM).* Technical Report 1235, National Institute of Standards and Technology, 1989.

[Albus 91] James S. Albus. *Outline of a Theory of Intelligence.* IEEE Transactions on Systems, Man and Cybernetics, vol. 21, no. 3, pages 473–509, May 1991.

[Arkin 87] Ronald C. Arkin. *Motor Schema based Navigation for a Mobile Robot.* In IEEE International Conference on Robotics and Automation, volume 1, pages 264–271, Raleigh, North Carolina, USA, March 1987.

[Badreddin 91] Essam Badreddin. *Recursive Behavior-Based Architecture for Mobile Robots.* Robotics and Autonomous Systems, vol. 8, no. 3, pages 165–176, 1991.

[Baroni 95] P. Baroni, G. Guida, S. Mussi & A. Vetturi. *A Distributed Architecture for Control of Autonomous Mobile Robots.* In ICAR - International Conference on Advanced Robotics, volume 2, pages 869–877, Sant Feliu de Guixols, Catalonia, Spain, September 1995.

[Benveniste 91] Albert Benveniste & Gérard Berry. *The Synchronous Approach to Reactive and Real-Time Systems.* Proceedings of the IEEE, vol. 79, no. 9, pages 1270–1282, September 1991.

[Boussinot 91] Frédéric Boussinot & Robert de Simone. *The ESTEREL Language.* Proceedings of the IEEE, vol. 79, no. 9, pages 1293–1304, September 1991.

[Brooks 86] Rodney A. Brooks. *A Robust Layered Control System for a Mobile Robot.* IEEE Journal of Robotics and Automation, vol. RA-2, no. 1, pages 14–23, March 1986.

[Brooks 91a] Rodney A. Brooks. *Intelligence without Reason.* In International Joint Conference on Artificial Intelligence, volume 1, pages 569–595, Sydney, Australia, August 1991.

[Brooks 91b] Rodney A. Brooks. *Intelligence without Representation.* Artificial Intelligence Journal, vol. 47, no. 1-3, pages 139–160, January 1991.

[Brooks 91c] Rodney A. Brooks. *New Approaches to Robotics.* Science, vol. 253, pages 1227–1232, September 1991.

[Camargo 91] Rogério Ferraz de Camargo. *Architecture matérielle et logicielle pour le contrôle d'exécution d'un robot mobile autonome.* Thèse de doctorat de l'université paul sabatier (ups), Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS/CNRS), July 1991. Thèse UPS 949, Rapport LAAS 91272.

[Chatila 92] Raja Chatila, Rachid Alami, Bernard Degallaix & Hervé Laruelle. *Integrated Planning and Execution of Autonomous Robot Actions.* In IEEE International Conference on Robotics and Automation, volume 3, pages 2689–2695, Nice, France, May 1992.

[Chatila 95] Raja Chatila. *Deliberation and Reactivity in Autonomous Mobile Robots.* Robotics and Autonomous Systems, vol. 16, pages 197–211, December 1995.

[CPN Group 96] CPN Group. *Design/CPN Home Page.* Page WEB, University of Aarhus, Denmark, 1996. URL: http://www.daimi.aau.dk/designCPN/.

[Fayek 93] Reda E. Fayek, Ramiro Liscano & Gerald M. Karam. *A System Architecture for a Mobile Robot Based on Activities and a Blackboard Control Unit.* In IEEE International Conference on Robotics and Automation, volume 2, pages 267–274, Atlanta, Georgia, USA, May 1993.

[Fleury 94] Sara Fleury, Matthieu Herrb & Raja Chatila. *Design of a Modular Architecture for Autonomous Robots.* In IEEE International Conference on Robotics and Automation, volume 4, pages 3508–3513, San Diego, California, USA, May 1994.

[Fleury 95] Sara Fleury, Philippe Souères, Jean-Paul Laumond & Raja Chatila. *Primitives for Smoothing Mobile Robot Trajectorie.* IEEE Transactions on Robotics and Automation, vol. 11, no. 3, pages 441–448, June 1995.

[Fleury 96] Sara Fleury. *Architecture de Contrôle Distribuée pour Robots Mobiles Autonomes: Principes, Conception et Applications.* Thèse de doctorat de l'université paul sabatier (ups), Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS/CNRS), February 1996. Thèse UPS 2258, Rapport LAAS 96156.

[Georgeff 86] M. P. Georgeff & A. L. Lansky. *Procedural Knowledge.* Proceedings of the IEEE, vol. 74, no. 10, pages 1383–1398, October 1986.

[Ghallab 88] Malik Ghallab & Hervé Philippe. *A Compiler for Real-Time Knowledge-Based Systems.* In IEEE International Workshop on Artificial Intelligence for Industrial Applications, Hitachi City, Japan, May 1988.

[Ghallab 89] Malik Ghallab & Amine Mounir Alaoui. *Managing Efficiently Temporal Relations through Indexed Spanning Trees.* In International Joint Conference on Artificial Intelligence, volume 2, Detroit, Michigan, USA, August 1989.

[Gouyon 95] Jean-Paul Gouyon. *KHEOPS User's Guide.* LAAS/CNRS, Toulouse, France, 1995. Rapport LAAS 94503.

[Hasemann 95] Jörg-Michael Hasemann. *Robot Control Architectures – Application Requirements, Approaches, and Technologies.* In XIV Intelligent Robots and Computer Vision: Algorithms, Techniques, Active Vision, Materials Handling (part of the SPIE's Photonics East Symposium), Philadelphia, Pennsylvania, USA, October 1995.

[Hayes-Roth 85] Barbara Hayes-Roth. *A Blackboard Architecture for Control.* Artificial Inteligence, vol. 26, no. 3, pages 251–321, July 1985.

[Ingrand 92] François Félix Ingrand, Michael P. Georgeff & Anand S. Rao. *An Architecture for Real-Time Reasoning and System Control.* IEEE Expert, vol. 7, no. 6, pages 34–44, December 1992.

[Ingrand 94] Fran cois Félix Ingrand. *C-PRS Development Environment Manual.* ACS Technologies, BP 556 – 31674 Labege – France, 1994.

[Ingrand 95] Fran cois Félix Ingrand, Raja Chatila, Rachid Alami & Frédéric Robert. *Embedded Control of Autonomous Robots using Procedural Reasoning.* In ICAR - International Conference on Advanced Robotics, volume 2, pages 855–861, Sant Feliu de Guixols, Catalonia, Spain, September 1995.

[Ingrand 96] Fran cois Félix Ingrand, Raja Chatila, Rachid Alami & Frédéric Robert. *PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots.* In IEEE International Conference on Robotics and Automation, volume 1, pages 43–49, Minneapolis, Minnesota, USA, April 1996.

[Jensen 90] Kurt Jensen. *Coloured Petri Nets: a High Level Language for System Design and Analysis.* In G. Rozenberg, editeur, Advances in Petri Nets, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer-Verlag, 1990.

[Jensen 94] Kurt Jensen. *An Introduction to the Theoretical Aspects of Coloured Petri Nets.* In J.W. de Bakker, W.-P. de Roever & G. Rozenberg, editeurs, A Decade of Concurrency, volume 803 of *Lecture Notes in Computer Science*, pages 230–272. Springer-Verlag, 1994.

[Kaelbling 88] Leslie Pack Kaelbling & Nathan J. Wilson. *Rex Programmer's Manual.* SRI International, Menlo Park, California, USA, 1988. Technical Report 381R.

[Kaestner 92] Celso A. Alves Kaestner. *Systèmes IA Temps-Réel: L'Apport de l'Approche Synchrone aux Systèmes à Base de Règles.* Rapport LAAS 92221, LAAS-CNRS, 1992.

[Khatib 86] Oussama Khatib. *Real Time Obstacle Avoidance for Manipulators and Mobile Robots.* International Journal of Robotics Research, vol. 1, no. 5, 1986.

[Khatib 95] Maher Khatib & Raja Chatila. *An Extended Potential Field Approach for Mobile Robot Sensor-based Motions.* In Intelligent Autonomous Systems (IAS'4), Karlsruhe, Germany, 1995. Rapport LAAS 95094.

[Lacroix 94] Simon Lacroix, Raja Chatila, Sara Fleury, Matthieu Herrb & Thierry Siméon. *Autonomous Navigation in Outdoor Environment: Adaptive Approach and Experiment.* In IEEE International Conference on Robotics and Automation, volume 1, pages 426–432, San Diego, California, USA, May 1994.

[Laumond 94] Jean-Paul Laumond, Paul E. Jacobs, Michel Taïx & Richard M. Murray. *A Motion Planner for Nonholonomic Mobile Robots.* IEEE Transactions on Robotics and Automation, vol. 10, no. 5, pages 577–593, October 1994.

[Lee 94] Sunro Lee & Robert M. O'Keefe. *Developing a Strategy for Expert System Verification and Validation.* IEEE Transactions on Systems, Man and Cybernetics, vol. 24, no. 4, pages 643–655, April 1994.

[Lewis 93] M. Anthony Lewis, Andrew H. Fagg & George A. Bekey. *The USC Autonomous Flying Vehicle: An Experiment in Real-Time Behavior-Based Control.* In IEEE International Conference on Robotics and Automation, volume 2, pages 422–429, Atlanta, Georgia, USA, May 1993.

[Liu 91] N. K. Liu & T. Dillon. *An Approach Towards the Verification of Expert Systems using Numerical Petri Nets.* International Journal of Intelligent Systems, vol. 6, pages 255–276, 1991.

[Medeiros 96a] Adelardo A. D. Medeiros & Raja Chatila. *Execution Control of an Autonomous Mobile Robots.* In SIRS - International Symposium on Intelligent Robotic Systems, pages 87–94, Lisboa, Portugal, July 1996

[Medeiros 96b] Adelardo A. D. Medeiros, Raja Chatila & Sara Fleury. *Specification and Validation of a Control Architecture for Autonomous Mobile Robots.* In IROS - IEEE International Conference on Intelligent Robots and Systems, pages 162–169, Osaka, Japan, November 1996.

[Murata 89] Tadao Murata. *Petri Nets: Properties, Analyses and Applications.* Proceedings of the IEEE, vol. 77, no. 4, pages 541–580, April 1989.

[Payton 86] David W. Payton. *An Architecture for Reflexive Autonomous Vehicle Control.* In IEEE International Conference on Robotics and Automation, volume 3, pages 1838–1845, San Francisco, California, USA, April 1986.

[Philippe 89] Hervé Philippe. *Algorithmes pour la Compilation de Bases de Connaissances en Logique Propositionnelle et du Premier Ordre: les Systèmes KHEOPS et CLOPS.* Thèse de doctorat de l'université paul sabatier (ups), Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS/CNRS), May 1989. Thèse UPS 31998, Rapport LAAS 89122.

[Schneider 95] Stanley A. Schneider, Vincent W. Chen & Gerardo Pardo-Castellote. *The ControlShell Component-based Real-Time Programming System.* In IEEE International Conference on Robotics and Automation, volume 3, pages 2381–2388, Nagoya, Aichi, Japan, May 1995.

[Simmons 92] Reid G. Simmons. *Monitoring and Error Recovery for Autonomous Wlaking.* In IROS - IEEE International Conference on Intelligent Robots and Systems, volume 2, pages 1407–1412, Raleigh, North Carolina, USA, July 1992.

[Simmons 94] Reid G. Simmons. *Structured Control for Autonomous Robots.* IEEE Transactions on Robotics and Automation, vol. 10, no. 1, pages 34–43, February 1994.

[Simon 93] Daniel Simon, Bernard Espiau, E. Castillo & Kapellos Konstantinos. *Computer-aided Design of a Generic Robot Controller Handling Reactivity and Real-time Control Issues.* IEEE Transactions on Control Systems Technology, vol. 1, no. 4, December 1993.

[Stewart 95] David B. Stewart & P. K. Khosla. *Rapid Software Development for Robotics Applications using Component-based Real-Time Software.* In IROS - IEEE International Conference on Intelligent Robots and Systems, Pittsburgh, PA, USA, August 1995.

[Thorpe 95] C. E. Thorpe & M. Hebert. *Mobile Robotics: Perspectives and Realities.* In ICAR - International Conference on Advanced Robotics, volume 1, pages 497–506, Sant Feliu de Guixols, Catalonia, Spain, September 1995.

[Tigli 93] J. Y. Tigli, M. Occello, M. C. Thomas. *Toward a New Intelligent Reactive Controller for Autonomous Mobile Robots.* In IEEE International Conference on Robotics and Automation, volume 3, pages 249–254, Atlanta, Georgia, USA, [illegible] 1993.

Contrôle d'exécution pour robots mobiles autonomes: architecture, spécification et validation

Le travail présenté dans le mémoire traite des problèmes liés au contrôle d'exécution des actions des robots mobiles autonomes. Une première partie présente l'architecture de contrôle globale et la compare à d'autres approches. On décrit les niveaux hiérarchiques qui la constituent et leurs rôles dans le fonctionnement du système. Le niveau inférieur, composé d'un ensemble de modules, rassemble les fonctions de perception, de modélisation et d'action du système.
La seconde partie présente le niveau exécutif. L'exécutif doit suivre l'exécution des fonctions, résoudre les conflits entre modules, accomplir certaines actions réflexes et maintenir une information sur l'utilisation des ressources non partageables du robot. Il peut être vu comme un ensemble d'automates, qui interagissent et changent d'état selon les requêtes qui arrivent du niveau supérieur et les répliques qui proviennent des modules.
La mise en oeuvre de l'exécutif utilise le système à base de règles KHEOPS. La compilation faite par KHEOPS permet, à partir d'un ensemble de variables d'entrée et de sortie et des règles qui les relient, d'obtenir un arbre de décision équivalent et de profondeur connue, ce qui garantit un temps d'exécution borné pour l'exécutif. La compilation permet aussi de garantir certaines propriétés logiques des automates mis en place.
La troisième partie présente les relations entre le niveau fonctionnel (modules et exécutif) et la couche immédiatement supérieure, le niveau tâche. Ce niveau est basé sur le système PRS, qui transforme des tâches de haut niveau d'abstraction en procédures d'actions reconnues par le niveau fonctionnel et surveille leur exécution. Le mémoire présente une équivalence entre un sous-ensemble de PRS et les réseaux de Pétri colorés, ce qui permet de faire une vérification du niveau tâche quand l'équivalence existe.
Enfin, on présente quelques résultats de la mise en oeuvre expérimentale de ces travaux avec le robot Hilare 2.

Mots clefs: Robotique mobile, Contrôle d'exécution, Architecture de contrôle, Spécification, Vérification, Systèmes temps-réel, Systèmes à base de règles, Réseaux de Pétri

Execution control for autonomous mobile robots: architecture, specification and validation

The work presented in the thesis deals with problems concerning the execution control of autonomous mobile robots. The first part of the work presents the global control architecture and compares it to other approaches. We describe the hierarchical levels which it is composed of and each of their roles in the system's functioning. The lower level, composed of several modules, combines the perception, modelisation and action functions of the system.
The second part presents the executive level. The executive must follow the execution of the functions, resolve the conflicts between modules, carry out certain reflex actions and maintain information concerning the use of the robot resources that cannot be shared. It can be seen as a group of automatons, that interact and change states according to the requests arriving from the higher level and the answers coming from the modules.
The implementation of the executive uses the rule-based KHEOPS system. KHEOPS' compilation, using a set of in-going and out-going variables and the rules that bind them, produces an equivalent decision tree whose depth is known, which guarantees that the execution of the executive will be bounded in time. The compilation also guarantees certain logical characteristics of the automatons used.
The third part presents the relations between the functional level (modules and executive) and the the level just above: the task level. This level is based on the PRS system, which transforms high abstraction-level tasks into action procedures that are recognised by the functional level, and supervises their execution. The thesis presents an equivalence between a PRS subset and coloured Petri nets, which enables checking at the task level, when the equivalence exists.
To conclude, we present some results regarding the experimental implementation of this work with the Hilare 2 robot.

Key words: Mobile robotics, Execution control, Control architecture, Specification, Verification, Real-time systems, Rule-based systems, Petri nets