Formalisation et intégration en vision par ordinateur temps réel

Soraya Arias

UNIVERSITÉ DE NICE – SOPHIA ANTIPOLIS
École Doctorale Sciences Pour l'Ingénieur

THÈSE

préparée à

l'Institut National de Recherche en Informatique et Automatique

(Unité de Sophia Antipolis)

présentée pour obtenir le titre de

Docteur ès Sciences

Discipline : Sciences pour l'Ingénieur

par

Soraya ARIAS

Formalisation & Intégration en Vision par Ordinateur Temps Réel

Soutenance le 16 Décembre 1999 à 14h00, devant le jury composé de :

Rapporteurs :	Augustin Lux	Professeur à l'Institut National Polytechnique de Grenoble
	Thomas Henderson	Professeur à l'Université d'Utah
Examinateurs :	Pierre Bernhard	Président, Professeur à l'Université de Nice-Sophia Antipolis
	Ève Coste-Manière	Directeur, Chargée de Recherche à l'INRIA Sophia
	Thierry Viéville	Directeur, Chargé de Recherche à l'INRIA Sophia
	Bernard Espiau	Examinateur, Directeur de Recherche à l'INRIA Grenoble
	Gérard Giraudon	Examinateur, Directeur de Recherche à l'INRIA Sophia

à l'Institut National de Recherche en Informatique et Automatique

Mis en page avec la classe thloria.

Résumé : Le domaine de la vision par ordinateur a atteint un degré de maturité qui lui permet d'envisager, au delà de la mise en oeuvre d'algorithmes utilisés au coup par coup ou dans un asservissement, la construction d'applications complexes intégrant différents aspects (analyse de scène, décision, traitements temps réel, supervision). Cette complexité se répercute à tous les niveaux du cycle de développement de ces applications (conception, validation et implantation).
Ce travail propose donc une méthodologie de conception et des outils effectifs pour la mise en oeuvre d'applications de vision temps réel.

La méthodologie de conception proposée exige un découpage fonctionnel des traitements en tâches élémentaires, puis l'organisation de ces tâches de manière logique pour construire l'application. Ces tâches élémentaires sont appelées des TÂCHES VISION. Elles se présentent sous la forme d'une boucle de calculs temps réel, paramétrée, contrôlée de manière logique et pouvant agir sur un capteur visuel.

L'environnement ORCCAD/MAESTRO, dédié initialement à la robotique, offre des caractéristiques intéressantes pour satisfaire aux besoins de cette méthodologie. Il offre en particulier des outils formels de validation de la partie liée au contrôle logique et permet la gestion rigoureuse des aspects temps réel. Afin de tirer le meilleur parti de cet environnement pour le développement d'applications de vision, nous avons dû ajouter ou enrichir certaines de ses fonctionnalités. Les modifications concernent notamment un mécanisme de paramétrage dynamique des calculs, un mécanisme de communication assurant l'interopérabilité du système et un mécanisme de génération automatique d'interfaces de supervision.

L'utilisation de cette méthodologie et de l'environnement ORCCAD étendu est illustrée à l'aide d'un prototype d'application de suivi de cible, dans un contexte d'assistance aux personnes âgées.

Mots clés : Applications de vision, Temps Réel, TÂCHE VISION, Méthodologie de programmation, ORCCAD, MAESTRO, Environnement de programmation.

Abstract : Advances in Computer Vision are making it possible to move forward from the design of algorithms used punctually or in visual servoing, to the conception of complex applications containing aspects such as scene analysis, decision making, supervision and real-time processing. This increased complexity reflects upon all the levels of the development process of these applications, namely the design, validation and implementation.
This work proposes a design framework for the implementation of real time vision applications, in addition to a variety of adequate tools.

The proposed framework is based on a functional structuring of the required processing into elementary tasks, followed by their logical composition in a order to build the vision application. These elementary entities are called VISION TASKS. They are seen as logically controlled, parametrized real time processing loops capable of controlling a visual sensor.

The ORCCAD/MAESTRO environment, developed initially for robotics control, exhibits interesting characteristics for its use in the proposed framework. Specifically, it enables the use of tools for logical control validation and allows efficient management of the real time resources. However, and in order to draw full benefit from using this environment for developing vision applications, we expand its functionality. Major modifications encompass an adaptive dynamic parametrization mechanism, a communication scheme that ensures the interoperability of the system and automatic generation of supervision interfaces.

An application of the proposed methodology and of the use of the modified ORCCAD environment are presented for a typical tracking application in a context of assistance to elderly persons.

Keywords : Real-Time, Vision Applications, Vision Task Formalism, Programming Methodology, ORCCAD, MAESTRO, Programming Environnement.

Remerciements

Les travaux présentés dans cette thèse ont été effectués au sein des équipes Icare et CHiR de l'INRIA Sophia Antipolis.

Je tiens à remercier Pierre Bernhard, professeur à l'Université Nice-Sophia Antipolis pour avoir accepter de juger ce travail et de présider le jury de soutenance.

Je remercie pour l'intérêt qu'ils ont porté à ce travail en acceptant d'en être les rapporteurs :

- Thomas Henderson, professeur à l'université d'Utah. Merci d'être venu de si loin pour assister à la soutenance.
- Augustin Lux, professeur à INPG Grenoble. Merci pour les discussions que nous avons eues autour du manuscrit.

Je tiens à remercier Bernard Espiau, directeur de recherche à l'INRIA Grenoble et Gérard Giraudon, directeur de recherche à l'INRIA Sophia, pour l'intérêt qu'ils ont manifesté pour ce travail au cours de ces trois années et pour avoir accepté de le juger.

Enfin, merci à Ève Coste-Manière et à Thierry Viéville, chargés de recherche à l'INRIA pour l'attention avec laquelle ils ont suivi ces travaux. Ainsi qu'à l'équipe Icare au sein de laquelle a débuté cette thèse. Et un grand merci particulier à Roger Pissard-Gibollet, et à Konstantin Kapellos, grands maîtres ès ORCCAD, qui ont su partager leur grand savoir "Orccadien" avec le petit disciple que je suis. Sans oublier, Nicolas Turro, le "Maestro", qui a eu la pénible mission de supporter l'odieux co-bureau que je suis.

Mais je ne saurais conclure sans remercier très chaleureusement toutes les personnes au sein de l'unité de recherche de l'INRIA Sophia, qui ont contribué, à des titres divers, au bon déroulement de cette thèse. Grand merci aux filles de la "doc", au Semir, à l'AGOS (et tout particulièrement Gisèle), aux secrétaires et surtout aux doctorants avec lesquels j'ai partagé quelques instants d'humanité au cours de ces trois années.

Table des matières

Table des figures

Chapitre 1

Introduction

Dans cette thèse nous proposons une méthodologie originale pour la conception et le développement d'applications de vision en temps réel. Dans ces applications, on cherche à appliquer de façon automatique ou interactive les techniques classiques du domaine de la vision et du traitement de séquences d'images, à un flot continu d'images, provenant par exemple d'une caméra. En plus des techniques issues du domaine de la vision, ces applications font donc aussi appel aux techniques issues du domaine de l'automatique et du temps réel ansi qu'à celles issues du domaine de l'intelligence artificielle. La conception et le développement de telles applications de vision en temps réel se heurte donc au difficile problème de l'intégration de l'ensemble de ces techniques.

Si les modules algorithmiques ont été intégrés "à la main" au sein des systèmes embarqués comprenant de la vision, au niveau théorique, ce problème d'intégration est un thème relativement neuf, car il ne se posait, jusqu'à ces dernières années, que de façon ponctuelle ou était résolu au cas par cas, sans soucis d'adopter une méthodologie adaptée. En effet, ce type d'applications était limité car elles exigeaient une puissance de calcul importante, qui ne pouvait être délivrée que par des calculateurs, alors, très coûteux. Désormais, l'augmentation constante de la puissance de calcul des composants de grande série permet dorénavant d'envisager des équipements capables de supporter ce type d'applications, ceci à un coût abordable. De tels équipements deviennent indispensables dans de nombreux contextes : domotique, assistance, sécurité, robotique, contrôle de qualité, loisirs, etc.

Dans ce travail, nous nous sommes plus particulièrement intéressés aux applications de vision lié au "monitoring" vidéo, en particulier, nous avons étudié une application de "surveillance" de personne invalide, la "mamy-sitter". Globalement, de telles applications requièrent la coopération de plusieurs sous-systèmes : (i) un sous-système d'asservissement temps réel pour les données issues du capteur, (ii) un sous-système de raisonnement et décision pour analyser les résultats issus de la partie asservissement pour fournir une réponse qualitative au problème à résoudre par l'application, et, (iii) un sous-système de supervision de l'ensemble, au niveau duquelle des interventions humaines peuvent être nécessaires. Le contexte dans lequel on prévoit de placer ces applications imposent des contraintes spécifiques en terme de temps d'exécution, de robustesse et de sûreté de fonctionnement.

Par conséquent, pour répondre à l'ensemble de ces besoins, il nous paraît indispensable de disposer d'**une méthodologie et des outils correspondant** afin de simplifier la conception et la mise en œuvre d'applications de vision temps réel, et assurer automatiquement le

respect des besoins en temps réel, robustesse et sûreté.

De nombreux travaux ont tenté d'apporter une réponse à certains de ces problèmes. Par exemple, des méthodes et des techniques de parallélisation des traitements de visions ont été étudiées afin de garantir le respect des contraintes temporelles [Lavarenne 91a, Serot 99].
D'autres travaux se sont intéressés à la partie raisonnement de l'applicationen proposant des architectures basées sur des techniques inspirées du domaine de l'Intelligence Artificielle (telles que système expert, interpréteur de règles). Ces architectures permettent de déterminer dynamiquement et par des langages de haut niveau les différents traitements à enchaîner par une application.

Toutefois, nombre de ces travaux ne proposent pas de méthodologie précise pour faciliter la conception des applications et pour permettre de valider de manière formelle le respect des contraintes liées à la sûreté de fonctionnement, la robustesse et le respect des contraintes temporelles. Pourtant, cela nous semble fondamentale.

En conséquence, nous avons choisi de traiter cette carence méthodologique en proposant une approche *orientée tâche*. Ceci nous écarte, dans un premier temps, des approches fondées sur les techniques de l'Intelligence Artificielle, dont l'intégration dans notre formalisme pourra faire l'objet d'études ultérieures.
Ainsi, notre contribution consiste à *formaliser les actions de vision* qui composent les applications de type monitoring vidéo en fournissant un "fil directeur" méthodologique à suivre. Pour ce faire, nous proposons dans cette thèse :

- une définition formelle de l'action élémentaire de vision temps réel, la *Tâche Vision*. Cette définition fournit une "brique" à partir de laquelle on construit une application de vision, robuste, et dont la sûreté de fonctionnement peut être validée de manière formelle au niveau logique. Cette "brique" met l'accent sur la dissociation entre le traitement du flot continu de données par une partie algorithmique et le traitement du contrôle de ce flot.
- une méthodologie qui allie spécification et validation au niveau de la logique de l'application et implantation avec la gestion transparente pour l'utilisateur des contraintes temporelles.

Pour mettre en œuvre cette méthodologie, nous utilisons un environnement déjà existant, dédié aux applications du domaine robotique : le système ORCCAD/MAESTRO [Team 98, Turro 99]. Ce système se base sur une architecture hiérarchique de spécifications dans laquelle notre méthodologie s'intègre.
L'utilisation de cet outil a conduit à une étude de l'adéquation de ce système aux besoin inhérents des applications de vision temps réel. Après avoir mis en évidence un certain nombre d'inadéquations entre notre méthodologie et ORCCAD, cette étude a débouché sur l'adaptation de cet environnement en ce qui concerne, notamment, la gestion des paramètres, le générateur de code temps réel et la mise en œuvre d'une architecture de communication entre les processus de raisonnement, entre autres, et les processus temps réel.

Pour rendre comptre de ce travail, ce document est structuré comme suit :

- Le chapitre 2, **Description des besoins à partir d'une étude de cas**, présente la problématique de ce travail à partir de la description d'une application type : la "mamy-sitter". Cette application va servir de "fil conducteur" tout au long de ce manuscrit.
- Le chapitre 3, **Analyse et Etat de l'art**, passe en revue les besoins de mise en œuvre des applications de vision et étudie comment ce problème est traité par différentes équipes intéréssées par ce sujet. Des outils proposés pour l'intégration de système sont également décrits.
- Le chapitre 4, **Formalisation de la partie traitements "Temps Réel"**, présente notre définition de l'entité élémentaire de spécification d'applications de vision temps réel. Pour ce faire, on analyse en détail les différentes actions de vision qui composent une application de suivi dynamique de cible dans une scène d'intérieur.
- Le chapitre 5, **L'architecture ORCCAD/MAESTRO**, présente l'environnement de programmation ORCCAD que nous avons choisi d'utiliser pour répondre aux besoins de spécification, validation et implantation de nos applications de vision. Cet environnement se fonde sur le formalisme de la TÂCHE ROBOT qui est très proche du formalisme que nous proposons au chapitre 4.
- Le chapitre 6, **Première mise en œuvre de la "mamy-sitter" avec ORCCAD/MAESTRO**, présente comment nous avons utilisé cet environnement pour implanter une première mise en œuvre de l'application de la "mamy-sitter", à l'aide du formalisme des TÂCHES VISION.
- Le chapitre 7, **Adéquation ORCCAD - Vision "Temps Réel"**, analyse l'adéquation de cet environnement aux prérogatives des actions et applications de vision. Ce chapitre propose ensuite les adaptations que nous avons réalisées sur le système pour palier aux insuffisances du système ORCCAD vis-à-vis de nos applications de vision. Il est important de noter que les modifications qui ont été apportées ne contreviennent pas à la fonctionnalité d'origine du système ORCCAD, à savoir mettre en œuvre des applications robotiques.

  Ce chapitre 7 souligne le manque de moyen de mise en œuvre à travers lequel les différentes entités composant une application peuvent s'échanger des informations en cours d'exécution.
- Aussi, le chapitre 8, **L'architecture de communication**, est consacré à l'architecture de communication et propose une solution pour inclure dans le système ORCCAD un canal de communication inter-processus.
- Enfin, le chapitre 9, **Conclusion**, clôture ce document en proposant un récapitulatif des principales idées présentées dans ce mémoire, ainsi que les avantages et les limites de notre approche, et ouvre des perspectives à ce travail.

Note : A la fin du document, le lecteur trouve un glossaire qui recense la définition des termes particuliers utilisés dans ce document, et auquel il peut se référer en cours de lecture.

Chapitre 2

Description des besoins à partir d'une étude de cas

Ce chapitre présente la problématique et les objectifs de cette thèse.
Pour ce faire, nous présentons un exemple typique d'application de vision temps réel. Il s'agit d'un application de "surveillance" vidéo permettant d'aider une personne invalide à être le plus autonome possible, que nous avons baptisée : la "mamy-sitter".

Cette présentation de l'application nous permet d'appréhender les différents aspects à gérer lorsque l'on veut mettre en œuvre une telle application, que ce soit au niveau des différents traitements à coordonner pour répondre aux buts de l'application, qu'au niveau des étapes de conception à suivre.

Les difficultés de conception que cela induit nous conduisent à proposer les objectifs de ce travail, à savoir : fournir une méthodologie et des outils adéquats pour la mise en œuvre d'applications de vision "temps réel".

2.1 Présentation d'une application type : "la mamy-sitter"

Le but de notre application est d'aider une personne invalide à être le plus autonome possible chez elle, et ce par une "surveillance" vidéo. En effet, il existe de nombreux aménagements qui permettent à ces personnes de pallier leurs handicaps dans la réalisation de tâches quotidiennes. Ces aménagements ne permettent toutefois pas de répondre aux besoins exceptionnels. Certains de ces besoins, tels qu'une chute, un accident, une agression requièrent une assistance urgente sans que la personne puisse demander explicitement de l'aide. Dans une telle situation d'urgence, une application de vidéo-surveillance est particulièrement utile, car elle permet de déclencher automatiquement l'alerte, à partir de l'observation et la position de la personne suivie.

Ce type d'application de vidéo-surveillance, qui exige le suivi de mouvements et de position et la reconnaissance de gestes spécifiques, est caractéristique des thèmes de recherche abordés actuellement en vision par ordinateur [Pinhanez 99].
Nous allons donc en décrire le fonctionnement de manière détaillé, afin d'en extraire les caractéristiques et d'analyser les contraintes et les besoins qu'elle génère.

2.2 Specification des besoins

Dans l'exemple que nous avons choisi, nous utilisons un système muni d'une caméra qui va veiller en continu sur l'activité de la personne et qui va détecter des situations singulières. Ce système va donc faire office de "mamy-sitter", le côté humain étant aussi à prendre en compte au niveau de l'ergonomie de ce mécanisme de surveillance, assurée par un système robotique de caméra.
Décrivons les spécifications de cette application, qui est illustrée par la figure 2.1.

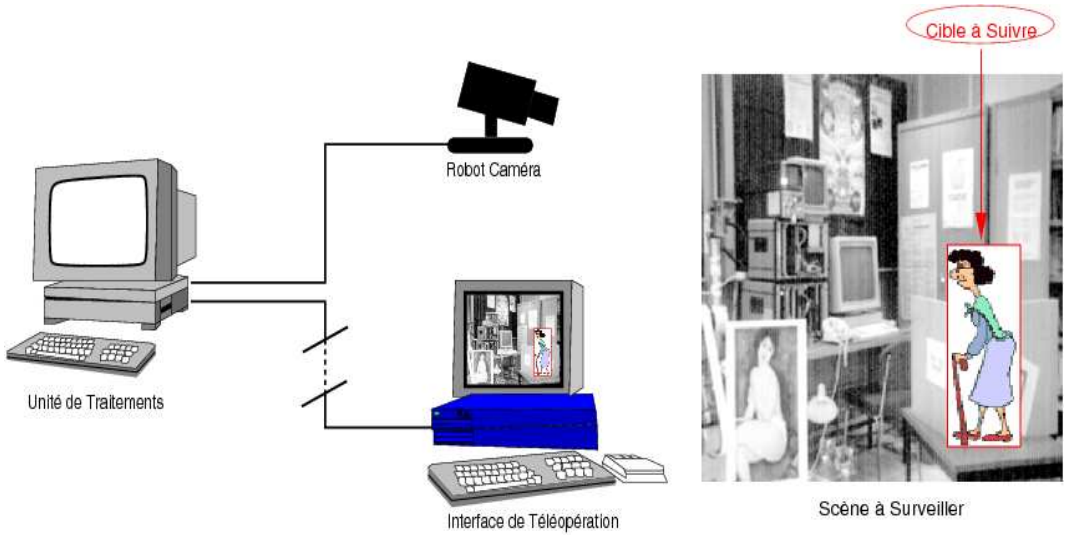


FIG. 2.1 – Présentation de l'application de la "mamy-sitter"

2.2.1 Fonctionnement nominal de l'application

Une caméra est installée dans une pièce, et une personne est présente dans la scène observée.

Le système exploite les images acquises afin de :

i. Suivre la personne lorsqu'elle se déplace.

ii. Analyser son comportement : i.e. interpréter ce qui se passe dans la scène par rapport à la conduite de la personne suivie. Par exemple,

`Où se trouve la personne ?`
`Est-elle étendue dans un lit / par terre ?`
`Est-elle dans une situation dangereuse ?`
Si une autre personne entre : `quelles sont les attitudes de ce personnage ?`
`L'attitude de ce personnage est-elle singulière ?`

iii. Déclencher des traitements spécifiques à une situation reconnue, en fonction des images de la caméra reçues à distance, on peut décider de :

`se focaliser sur une cible particulière;`
`se focaliser sur un endroit précis de la pièce;`
`déclencher un mécanisme d'alerte en cas d'urgence`

En d'autres termes, à distance, un téléopérateur peut changer la caméra de position (horizontale et verticale), varier le zoom, et réagir interactivement vis-à-vis des traitements et de la scène.

2.2.2 Fonctionnalités secondaires

En plus des actions définissant le fonctionnement nominal de l'application, on peut prévoir les actions suivantes :

i. Mise en veille : le système peut être mis en état de veille (i.e. arrêter le suivi et l'analyse de la scène) par le contrôle à distance ou lors de la reconnaissance dans la scène observée d'un geste particulier qui signifie l'arrêt du suivi et de l'analyse.
   En état de veille, le système peut passer à l'état de suivi soit par le contrôle à distance, soit par détection d'un mouvement, ou encore par reconnaissance d'un geste particulier.
   Un mécanisme de "chien de garde" assure que si le système tombe en panne (problème avec la caméra ou problème logiciel), le contrôle à distance est informé des raisons du dysfonctionnement.

ii. Contrôle et téléopération : en ce qui concerne les changements du système faits à distance (ou en téléopération), on dispose de réglages manuels pour modifier la position horizontale et verticale de la caméra, ainsi que celle du zoom : ceci permet de choisir un angle de vue particulier à distance pour l'observation de la scène.
   De plus, le téléopérateur dispose de mécanismes graphiques interactifs qui permettent de sélectionnner, sur l'image, la cible à suivre. On peut également envisager de pourvoir ce "poste de contrôle" de boutons pour réguler le système. Ces boutons sont associés à des informations susceptibles de modifier l'exécution des traitements et le résultat de l'analyse : par exemple, la luminosité, le nombre de cible maximum, la taille des cibles à detecter.

2.2.3 Contraintes de performance

Le "bon" fonctionnement du système se traduit par un faible taux d'erreurs au niveau de (i) la détection et le suivi de cible (ii) l'analyse du comportement de la cible. Cela implique une adaptation du suivi et de la détection à la dynamique de la scène observée et il faut prévoir la gestion de tout événement singulier au cours des traitements. En d'autres termes, l'application doit garantir les deux propriétés suivantes :

1. La validité : les spécifications énoncées pour l'application doivent être respectées ;
2. La robustesse : si une exception survient, le système doit rester dans un état stable.

De plus, la dynamique du système doit être respectée (en particulier vis-à-vis des contraintes temporelles) et la réactivité du système doit être assurée.

Les contraintes matérielles apparaissent à deux niveaux : au niveau du capteur visuel et au niveau de l'unité de calcul :

i. En ce qui concerne le capteur visuel, nous nous imposons un équipement tel que : une caméra robotique standard monoculaire couleur dotée d'un zoom. Elle est fixée sur une tourelle pour permettre des déplacements horizontaux et verticaux.

ii. Pour ce qui est de l'unité des traitements informatique, l'acquisition d'images et leur analyse sont gérées par une station de travail classique.

iii. Pour la partie contrôle à distance, on ne souhaite utiliser qu'une console classique. Elle va permette de visualiser les images issues de la scène et d'interagir avec le système.

Maintenant que nous avons décrit les spécifications de l'application, nous allons nous intéresser à sa mise en œuvre.

2.3 Conception - Mise en œuvre

2.3.1 Architecture générale

Pour mettre en œuvre l'application de la "mamy-sitter", nous devons définir les différents composants de l'application.
La figure 2.2 décrit de manière schématique la demarche de conception d'une telle application.

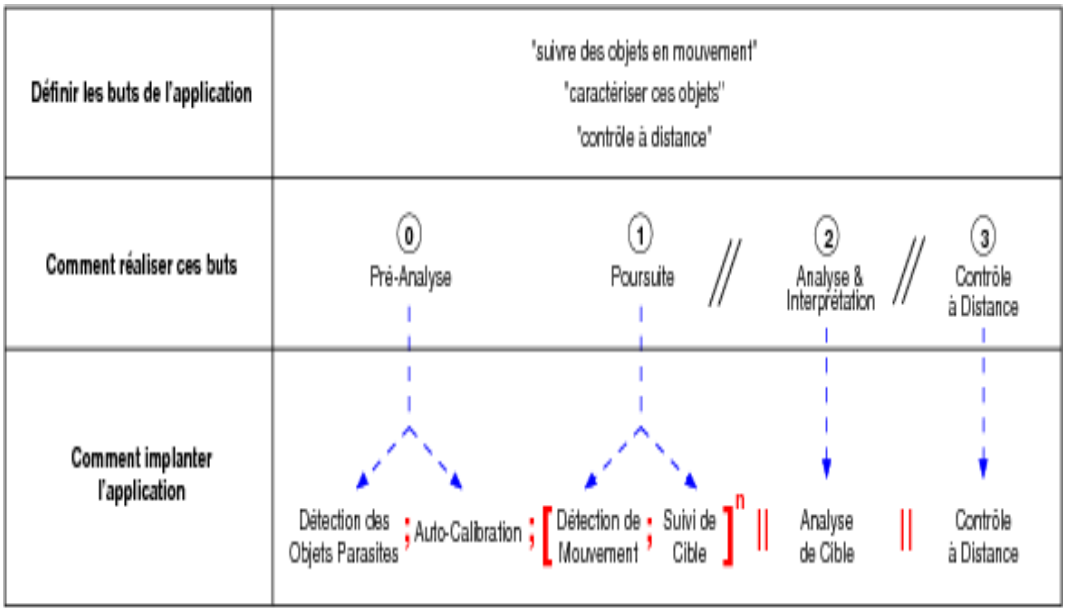


FIG. 2.2 – Les différentes étapes de développement de l'application

Tout d'abord, on définit les buts généraux que l'application doit satisfaire, à savoir, `suivre des objets en mouvement`, `les caractériser`, `contrôler à distance ces traitements`. Ensuite, on raffine ces buts en différentes phases (avant - la calibration- , pendant), on précise les besoins en terme de parallélisme, de séquencement, d'intervention extérieur. Enfin, on arrive aux entités élémentaires qui vont composer l'application, dont certaines sont des boucles de traitements temps réel. Par conséquent, intéressons-nous aux élements et aux procédés à utiliser pour réaliser l'application de la "mamy-sitter".

2.3.2 Description des phases de traitements

[Davis 97, Hager 96, Murray 94, Nordlund 96, Bremond 97] montrent que pour suivre une cible donnée dans une scène d'intérieur, les différentes phases de traitements suivantes sont nécessaires :

Calibrer la caméra et ses mouvements. Pour être capable de détecter des objets, le système doit passer par une phase préalable d'apprentissage dans laquelle il acquiert des informations sur la scène à observer : il détecte les objets qui pourraient interférer avec une analyse visuelle (scintillements d'un écran de télévision, mouvements d'un ventilateur, etc.), il repère les emplacements où la personne à assister est susceptible de rester immobile (un lit, un fauteuil, etc.), il modélise la topologie de la pièce (portes, murs, placards, etc.).
Les informations collectées lors de cette pré-analyse servent par la suite au niveau de l'analyse et du suivi. Cette pré-analyse peut se faire en deux étapes :

1. une première étape établit une carte grossière de l'environnement visuel de la scène ;
2. une deuxième étape de localisation d'objets peut être réalisée interactivement avec la personne à assister sur une mosaïque d'images.

La phase d'élaboration de la carte peut être réalisée au moment de la *calibration* du système.

Détecter et suivre des objets en mouvement. C'est la partie critique du système, car elle impose des contraintes sur la durée des traitements périodiques. Elle exige que les calculs soient cadencés en fonction de la dynamique de la cible à suivre (i.e. en *temps réel* [Gillies 95]) sur un flot de données important (suite de transformations sur des images).

Interpréter la scène. Pendant la phase de suivi, le système doit déterminer les propriétés d'une cible observée : trajectoire, taille [1]. En fonction des informations collectées lors de la phase d'apprentissage, le système peut déduire la position de la cible, et tenter de valider des scénarios de comportement [Chleq 99].

Contrôler à distance. Il doit permettre de téléopérer la caméra à distance pour modifier sa position, visualiser la scène observée et la cible en cours de suivi. Il doit permettre le réglage des paramètres de fonctionnement du système.
Pour ce faire, on peut envisager une interface homme-machine qui facilite cette téléopération.

2.4 Analyse

Maintenant que les phases de traitements ont été discutées, regardons les problèmes à résoudre pour leur mise en œuvre :

[1] La taille permet, par exemple, de détecter une position anormale de la cible (par exemple, une personne gisant sur le sol).

- **Gestion des traitements.** La première phase se fait à l'initialisation du système. Les trois suivantes sont à lancer conjointement, et correspondent à trois ensembles de traitements concurrents mais qui coopèrent en communiquant des informations nécessaires à leurs fonctionnements respectifs.
  Donc, le système doit assurer la cohésion entre ces deux phases quelques soient les événements non attendus qui pourraient survenir : gérer le respect des contraintes temporelles, ainsi qu'intercepter et traiter les exceptions se produisant lors de l'exécution. Étant donné que la sécurité des personnes dépend de la bonne exécution de cette étape temps réel, il faut pouvoir estimer et supprimer les effets de bord. Ainsi, une simple "boucle" de suivi n'a aucune chance de remplir ce cahier des charges : *la complexité afférente au respect des contraintes temps réel et à la gestion des exceptions devenant trop grande pour être gérer convenablement*.
- **Complexité des traitements.** Chaque tâche nécessite des algorithmes qui manipulent des séquences d'images et qui doivent s'exécuter périodiquement. Le résultat des traitements algorithmiques peut fournir : soit la commande à un capteur (par exemple, "déplacer la caméra horizontalement"), soit une information qualitative (par exemple, "la grand-mère est tombée") ou encore une information quantitative (par exemple, la taille d'un objet en mouvement).
- **Gestion du capteur.** Chaque tâche va faire appel à un capteur pour obtenir des images de la scène à observer. Par conséquent, ces ressources jouent un rôle critique dans la mise en œuvre de la tâche. Aussi, faudra-t-il contrôler les capteurs pour s'assurer que les images sont fournies de manière idoine et que les déplacements sont effectués correctement.
- **Gestion des dysfonctionnements.** Chaque tâche peut voir son fonctionnement nominal perturbé lorsqu'elle se trouve à gérer des situations inopinées : (i) impossible de démarrer la tâche, lorsque, par exemple, la caméra n'a pas été calibrée au préalable, (ii) impossible de fournir un résultat convenable, par exemple, dans le cas où la luminosité est trop faible, (iii) fournir un résultat singulier, par exemple, si la grand-mère tombe, une alarme doit être enclenchée. Ces différentes situations doivent être gérées de façon uniforme et donnent lieu à des décisions d'ordre logique. Ces décisions peuvent être prises en fonction d'un comportement particulier dicté par l'application ou la nature de la tâche à implanter.
- **Estimation et adaptation des résultats.** Si les images sont trop bruitées, la tâche implantant le suivi peut considérer des zones bruitées comme des régions en mouvement. L'influence de ce bruit peut être modifiée par un seuil, ou plus généralement peut être paramétrée. En fonction de la valeur donnée au paramètre, le résultat de la tâche peut varier. Par conséquent, il serait intéressant de pouvoir estimer la qualité d'un résultat et d'ajuster les paramètres afin de pouvoir améliorer le résultat.
- **Gestion des contraintes temporelles.** Certaines tâches sont plus critiques que d'autres vis-à-vis du respect des contraintes temporelles (par exemple, la détection et le suivi de cible). Ce respect influe fortement sur les performances de la tâche. Plus particulièrement, il faudrait pouvoir gérer tout dépassement d'une échéance de temps ou mieux encore, il faudrait pouvoir le prédire.

– **Besoins en communication.** Des processus[2], tels que l'interface homme-machine (notée IHM), qui sont extérieurs à la chaîne des traitements de vision, ont besoin de communiquer avec cette dernière pour échanger des informations numériques (par exemple, le résultat d'un traitement, une nouvelle valeur de paramètre) ou bien encore, des informations concernant le contrôle (par exemple, arrêt de l'application, relance de l'application, modification d'un paramètre).
Cette communication exige donc la définition d'un protocole de synchronisation et d'échange d'information.
– **Coordination des actions.** Une application complexe est composée de plusieurs tâches. Aussi, pour implanter une telle application, il faut pouvoir spécifier l'application comme un arrangement logique de compétences qui élude tous les détails relatifs à l'implantation. Par conséquent, il faut définir une structure qui permette d'encapsuler les compétences pour favoriser la modularité et la réutilisabilité des composants, et qui permette, de plus de faciliter le contrôle de ces composants à un plus haut niveau.

2.5 Conclusion

2.5.1 Récapitulatif

En résumé, la mise en œuvre d'applications de vision par ordinateur (de même type que celle présentée ci-dessus) requiert à la fois :

- des algorithmes de vision sophistiqués, qui font appel à des techniques issues de différentes approches propres à la vision par ordinateur (géométrie/calibration, interprétation de scène, mouvement, traitements d'images bas niveau) ;
- des processus décisionnels qui font appel le plus souvent à des techniques liées à l'Intelligence Artificielle ;
- des commandes robotiques (pour le contrôle du capteur visuel) ;
- des processus interactifs (tels qu'une IHM) ;
- l'estimation de résultats / l'adaptation de paramètres ;
- la gestion de contraintes liées au temps réel ;
- la gestion de communications / contrôle de processus diversifiés tels que des traitements périodiques, des commandes robotiques, des IHMs, des traitements décisionnels ;
- si possible des composants réutilisables.

Il est donc nécessaire de générer un formalisme rigoureux pour gérer correctement cette complexité.

[2] Attention, dans ce document, nous utilisons le terme *processus* pour désigner "une suite ordonnée d'opérations aboutissant à un résultat" [Robert 97], et non pas dans le sens de *processus* UNIX, qui désigne un programme en cours d'exécution sur un processeur, caractérisé par un contexte contenant les informations sur son état d'exécution (données, code, pile)

2.5.2 Objectif

Par rapport à cette problématique, l'objectif de ce mémoire est de :

Proposer une méthodologie intégrant des outils "formels" pour intégrer **spécifications**, **validations**, et **implantations** d'actions de vision.

Bien entendu, on ne va pas traiter tous les problèmes. Compte-tenu de l'état de l'art en la matière (voir le chapitre 3), des outils déjà disponibles ou au contraire des carences dans le domaine, il nous a paru important de s'intéresser, avant tout, à la partie dynamique du système. Plus précisément, notre but est de définir une entité élémentaire de vision sous contrainte temps réel, et de proposer un outil (ORCCAD [Borrelly 90]), et son adaptation au domaine de la vision temps réel en y intégrant toutes les spécificités, qui permette de spécifier, valider et implanter une application à partir de ces différentes entités élémentaires.

Ainsi, notre contribution ne s'intéresse pas directement aux aspects raisonnements ou décisionnels liés au domaine de l'Intelligence Artificielle.

Chapitre 3

Analyse et Etat de l'art

Au chapitre 2, nous avons présenté une application type de vision qui illustre la problématique à laquelle nous souhaitons donner une solution par ce travail.
Nous souhaitons fournir des méthodes et un environnement à travers lesquels on puisse spécifier, valider et implanter de manière systématique des applications de visions dont la caractéristique principale est qu'elles requièrent des besoins spécifiques en matière de temps réel, de robustesse et de sûreté de fonctionnement, car on veut que ces applications soient autonomes, ce qui supposent souvent que le code d'implantation soit embarqué.

Dans ce chapitre, nous analysons les applications de vision type qui intéressent ce travail, et donc nous avons présenté un exemple au chapitre 2.
Ces applications sont caractérisées par :

- une suite de traitements algorithmiques périodiques à partir d'une image issue d'un capteur visuel pour en extraire l'information nécessaire à l'application donnée. Ces traitements sont soumis à des contraintes temporelles particulières qui dépendent de l'environnement et de l'application.
- un processus de raisonnement ou de décision pour répondre à une question précise relative à la scène observée et à l'objectif de l'application (par exemple, une personne s'est déplacée vers la fenêtre de droite).
- un processus de supervision qui permet, par téléopération, de contrôler à distance les traitements algorithmiques et le processus de raisonnement.

La figure 3.1 synthétise l'organisation générale des applications de vision que nous étudions.

Pour chacune de ces caractéristiques, nous allons étudier leurs besoins de spécification et voir ce qui a déjà été fait dans la littérature. Ensuite, nous analysons ce qui est proposé pour intégrer ces éléments afin de composer des applications complexes. Nous nous intéressons tout particulièrement au respect des contraintes temporelles et la gestion de la dynamique de la scène.

3.1 Traitements temps réel des données

Dans ce paragraphe, nous allons étudier un des aspects qui composent les applications de vision qui nous intéressent : les traitements algorithmiques temps réel.

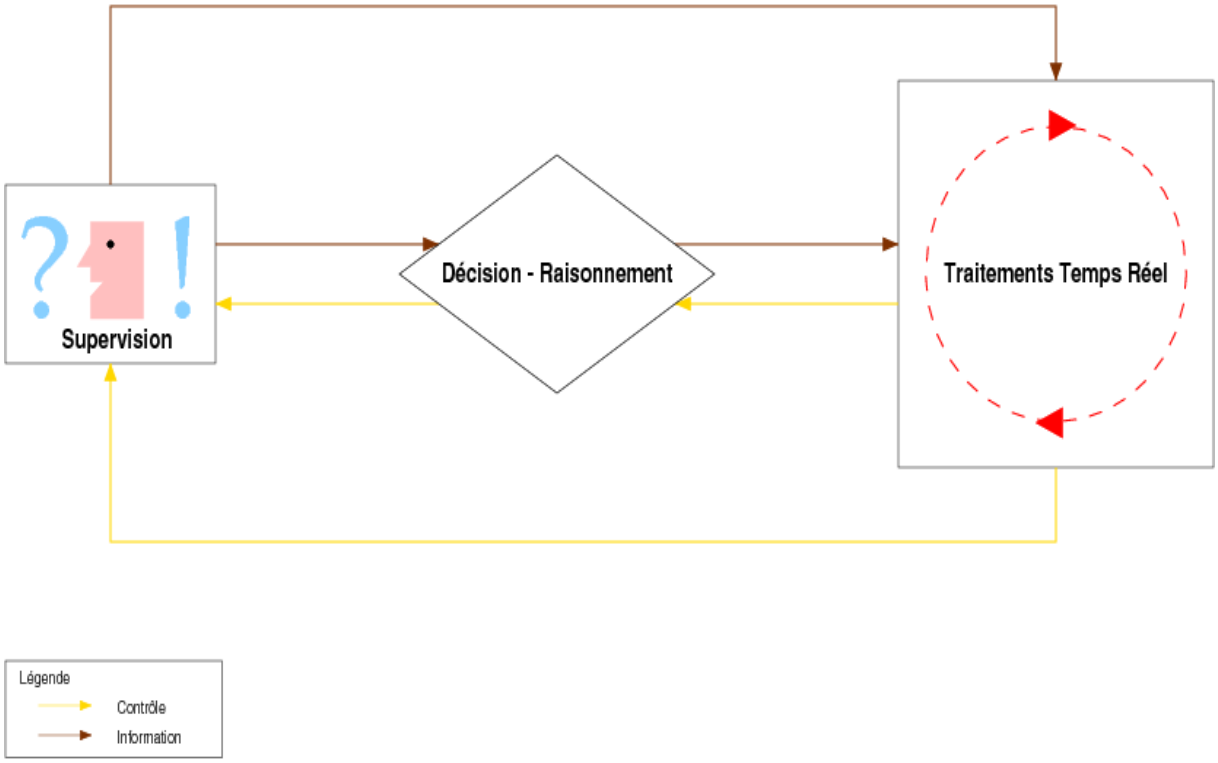


FIG. 3.1 – Structure des applications de vision dans le domaine considéré.

Nous allons nous pencher sur les caractéristiques des traitements de vision au niveau algorithmique, et au niveau de leur complexité. Ensuite, nous nous intéressons aux contraintes temporelles que doivent satisfaire ces algorithmes. En effet, étant donné que nous nous intéressons à des applications temps réel, il faut qu'au niveau des algorithmes on puisse garantir un temps de calcul qui soit compatible avec la cadence imposée par la scène observée (par exemple, la cadence vidéo, 25 images par seconde).

3.1.1 Les algorithmes

Dans le domaine du traitement d'images et plus généralement de la vision par ordinateur, les structures algorithmiques qui interviennent sont caractérisées par :

- des structures de données variées : par exemple, une matrice bi-dimensionnelle pour modéliser une image, une liste chaînée, un graphe d'attributs.
- des algorithmes mettant en jeu des traitements de différentes natures. Traditionnellement, on distingue trois classes d'abstraction : le bas, le moyen et le haut niveau.

  i. Le bas niveau

  Les traitements de bas niveau concernent essentiellement l'ensemble des opérations de pré-traitement et de transformation dont la caractéristiques première est de conserver la topologie bi-dimensionnelle de l'image. Ces traitements peuvent être vus comme des transformations $image \rightarrow image$.

  Une même séquence d'instruction est appliquée à chaque point de l'image dans le but de faire ressortir certaines caractéristiques.

  Exemples : le filtrage, les opérations différentielles, la rectification d'image, le calcul de la carte de disparités, le calcul du champ normal de mouvement, etc.

  Au niveau complexité algorithmique, le nombre d'opérations élémentaires mis en jeu dans ces traitements est fonction de la taille de l'image. Par conséquent, le temps d'exécution est proportionnel au volume des données.

  ii. Le moyen niveau

  Les traitements du moyen niveau concernent les opérations d'extraction de primitives images, c'est-à-dire d'information utile dans les images traitées par les processus bas niveau. Ces traitements manipulent conjointement des représentations bi-dimensionnelles et symboliques.

  L'objectif de cette étape intermédiaire est de diminuer le volume des données pour ne conserver que l'information utile.

  Exemples : la détection de coins, la segmentation en régions, la détection de contours, etc.

  Au niveau algorithmique, il est possible de prévoir le nombre et la taille des données à traiter. Ce nombre dépend du nombre de pixels de l'image ou d'un voisinage **et** du traitement lui-même (par exemple, itérer une opération jusqu'à ce que le résultat soit inférieur à un seuil fixé), il est lié aux paramètres de ce traitement (par exemple, la valeur seuil du nombre d'itération).

  Par conséquent, du point de vue temporel, on pourra au mieux estimer une borne supérieure pour le temps d'exécution.

  iii. Le haut niveau

  Les traitements de haut niveau vont exploiter les primitives issus du niveau inter-

médiaire pour interpréter l'image ou faire de la reconnaissance des formes. Ces traitements sont le plus souvent basés sur des techniques de mise en correspondance des primitives avec un modèle extrait d'une base de données. Exemples : suivi d'attributs, approximation polynômiale, opérateurs différentiels sur des courbes, reconstruction de courbes, etc.

Le tableau 3.1 récapitule les caractéristiques des différents type de traitements en images.

	Structure de données	Traitements	Complexité Θ [1]
Bas Niveau	matrice image	opérations locales au niveau pixel	$\Theta = K * \text{taille matrice}$
Moyen Niveau	liste de points graphe de relation	opérations itératives et récursives	variable borne supérieure envisageable
Haut Niveau	liste graphe	mise en correspondance avec des bases de données	variable

TAB. 3.1 – Caractéristiques des traitements d'images.

3.1.2 Les environnements de programmation

Pour traiter ces différents types d'algorithmes, un certain nombre d'environnements conçus comme des bibliothèques de fonctions ont été proposés. Nous allons présenter, dans ce qui suit, quelques exemples représentatifs de ces environnements.

CVIPTOOLS [Umbaugh 98]. L'environnement CVIPTOOLS (Computer Vision and Image Processing tools a été développé dans un but éducatif pour permettre à des non-spécialistes d'accéder facilement aux différentes opérations de traitements d'images et d'apprendre par l'exemple ce qu'est le traitement d'image par ordinateur. Mais il permet aussi de développer des applications de traitements d'images.

Il est composé de différentes bibliothèques de fonctions C regroupées par thème représentatif de l'état de l'art en traitement d'image (traitements arithmétiques, compression, extraction d'attributs visuels, histogrammes, filtrages, etc.). Il définit un objet image particulier, ainsi que d'autres structures de données particulières pour manipuler ces fonctions (comme un type particulier pour les booléens, des constantes prédéfinis, des formats d'image, des matrices, etc.).
Ces bibliothèques contiennent pour la plupart des traitements bas niveau, et moyen niveau. Elles sont régulièrement mis à jour pour supporter de nouveau types de traitements.

Il propose aussi un environnement graphique (en TCL) qui permet d'accéder facilement aux fonctions des bibliothèques de traitements sans avoir à programmer. Cet environnement graphique consiste en différentes fenêtres graphiques pourvu de menus qui permettent de façon interactive de charger une image, de lancer les différents types de traitements, de visualiser l'image résultante, de modifier les paramètres des traitements .

TARGETJR [TARGET WEB Page]. TARGETJR est un environnement pour l'interprétation d'images créé comme un outil adressé à la recherche en vision par ordinateur et comme une plateforme pour mettre en œuvre des applications d'interprétation d'images. Cet environnement permet d'échanger rapidement des algorithmiques entre différentes équipes.

TARGETJR se présente comme un ensemble de bibliothèques de classes C++ développées pour répondre au domaine de l'interprétation. Toutefois, l'accent est mis sur l'approche géométrique pour l'analyse d'image. Ces bibliothèques fournissent des méthodes depuis des algorithmes de bas niveau pour le traitement d'image avec la définition d'une structure particulière pour l'image jusqu'à des outils pour modéliser des objets réel et pour faire des opérations géométriques complexes (sur des objets 3D).

TARGETJR offre à l'utilisateur deux interfaces possibles :

1. une interface (menus, fenêtre de visualisation du résultat) avec lequel il peut utiliser directement la librairie d'algorithmes de traitement d'images ou de manipulation d'objets géométriques. Dans ce cas, l'utilisateur peut charger une image, appliquer directement un des algorithmes disponibles en cliquant sur un des choix des menus de traitement (manipulation géométrique, traitement d'images) et visualiser le résultat.
2. une interface (squelettes de programmes principaux, *callback*, *makefile*) pour réaliser et mettre au point ses propres algorithmes de traitements d'image ou de manipulation d'objets géométriques.
   Pour l'instant, TARGETJR ne supporte que les langages C et C++.

A l'origine, cet environnement provient des laboratoires de recherche de *General Electric* qui se sont intéressés à l'analyse d'images *rayon X*. Ainsi, les images sont considérés comme des *photographies* dans le sens où on ne peut visualiser et manipuler que des images dont les pixels sont à valeur positive. Ceci limite quelque peu (tout du moins pour l'instant) l'utilisation de TARGETJR.

Cet outil est fortement apparenté à IUE (Image Understanding Environnement) [Mundy 93], environnement de programmation développé sous l'impulsion du *DARPA*.
IUE a pour but de fournir une hiérarchie de classes C++, des bibliothèques d'algorithmes et des outils de visualisation et d'interface utilisateur, qui vise à couvrir les besoins des chercheurs travaillant en interprétation d'images.
Il est prévu, à moyen terme, que ces deux outils fusionnent en une plateforme logicielle de traitement d'image standard.

XVISION [Hager 96].

XVISION est un environnement de programmation pour la vision temps réel qui fournit une ensemble d'outils pour le suivi visuel d'attributs. Cet environnement a été conçu pour être flexible et très performant sur des stations de travail ou des PCS standard équipées de capteur visuel simple. Il est destiné au prototypage rapide d'applications utilisable pour l'enseignement et la recherche.

XVISION est élaboré autour d'un ensemble de classes C++, qui fournit des méthodes pour le suivi d'attributs et pour l'affichage d'image, pour la gestion du capteur vidéo. Il gère des traitements bas niveau et moyen niveau (avec, par exemple, la manipulation d'objet ligne ou coin). Cet environnement a été utilisé pour implanter, entre autres, une application [Hager 98] qui suit les yeux et la bouche d'une personne et qui superpose à l'image traquée

la figure d'un clown qui retranscrit la forme et la position des yeux et de la bouche. Les performances temporelles obtenues sont de $10ms$ pour chaque image traitée.

Conclusions. Ces différentes approches ne proposent pas de méthodologie de programmation particulière, dans le sens que l'utilisateur va programmer classiquement la partie traitement de vision de son application en utilisant au mieux les fonctions de bibliothèques. CVIPTOOLS ET TARGETJR proposent des environnements graphiques qui permettent l'utilisation interactive de ces fonctions de bibliothèques.
Cependant, aucune de ces bibliothèques ne tient compte explicitement d'une structure de contrôle associée aux algorithmes de traitement utilisés.

3.1.3 Discussion

Les applications de vision que nous considérons dans ce travail doivent s'exécuter suivant une cadence imposée par l'environnement, cette cadence dépend de la dynamique de la personne suivie. Nous travaillons dans le cadre d'applications dites *temps réel*, pour lesquelles le bon fonctionnement des applications ne dépend pas seulement de l'exactitude du résultat mais aussi du temps auquel ce résultat est produit.
Par conséquent, les traitements algorithmiques doivent satisfaire des contraintes temporelles particulières. Plus spécifiquement, ils doivent produire des informations en réaction avec les variations d'état de l'environnement. De plus, la latence des traitements doit être telle qu'elle est délimitée par la cadence imposée par l'environnement.

Or, nous avons vu au paragraphe 3.1.1 que les traitements en vision sont fortement dépendant de la quantité de données à traiter en entrée et et du type de traitement envisagé (par exemple, itérations).
Étudions les moyens qui permettent d'assurer le respect des contraintes temporelles, afin entre autres, de traiter des images plus importantes ou de travailler à des fréquences plus élevées.

- La parallélisation des traitements. Les processeurs séquentiels sont de plus en plus performants et résolvent rapidement des problèmes coûteux en temps d'exécution. Cependant, la capacité de calcul des processeurs, même si elle augmente, n'est pas infinie[2]. De plus, certains systèmes de traitements et d'analyse d'image nécessitent des puissances de calcul qu'aucun processeur ne peut fournir actuellement (par exemple, les techniques à base de recuit-simulé).
  Une solution intéressante pour faire face aux besoins de puissance de calcul est la parallélisation des traitements. Pour ce faire, on peut envisager notamment :
  - de paralléliser la séquence de traitements mis en jeu pour mettre en oeuvre la partie qui transforme les images en temps réel en utilisant la technique de pipeline.
  - de paralléliser les algorithmes de traitements.

  Le principe du pipeline s'applique sur des traitements de données qui peuvent être séquentialisés. Pour chaque traitement de la séquence, on affecte un processeur. Et

[2]La loi de Gordon Moore, l'un des fondateurs d'Intel, énonce que *le nombre de transistors intégré dans une puce doublera tous les deux ans*. Mais depuis 1992, la courbe de progression prévue par Moore s'est infléchie

chacun des processeurs fait un calcul différent sur une donnée qui passe d'un processeur à un autre, tout comme sur une chaîne de montage.
Avec cette technique, on ne diminue pas le temps de calcul de bout en bout de la séquence de traitements (la latence), mais en revanche, on augmente le débit de la chaîne de traitements globale (pour se rapprocher par exemple de la cadence vidéo).
Un algorithme parallèle se caractérise le plus souvent par un ensemble de séquences d'instruction. dont chacune est allouée à un processeur différent. On introduit au sein de ces séquences des instructions de communication qui ont pour but de gérer les dépendances de données entre les instructions s'exécutant sur des processeurs différents.
Lorsque l'on va paralléliser un algorithme, on cherche à décomposer un traitement global en un ensemble d'actions s'exécutant sur des processeurs différents. Cette décomposition peut s'effectuer soit au niveau des opérations, soit au niveau des données.
L'identification du type de parallélisme est fortement lié à la notion de **granularité** des calculs et des données. Cette granularité indique le volume de travail effectué par un processeur indépendamment des autres. Ainsi, on distingue :
– un parallélisme *à grain fin* : où le volume correspond à quelques instructions élémentaires, telles que des additions, des comparaisons et/ou à quelques données élémentaires telles que le pixel.
– un parallélisme *à grain moyen et fort* : où on regroupe et on exécute séquentiellement des instructions et/ou des données.

[Weems 91] ont montré que les traitements en vision se caractérisent par différents types de parallélisme. Les traitements bas niveau sont bien adaptés à une parallélisation à *grain fin* et peuvent être gérés par des machines de type *SIMD* ou *Single Instruction Multiple Data*. Les traitements moyen niveau et haut niveau sont plutôt adaptés à une parallélisation à *grain moyen et fort* et peuvent être gérés par des machines de type *MIMD* ou *Multiple Instruction Multiple Data*, du fait des structures de données plus complexes et des calculs plus complexes.
En effet, sur les machines *SIMD* ou encore machines vectorielles, la même instruction est traitée simultanément sur plusieurs données par les différents processeurs de la machine. Les machines *MIMD*, quant à elles, permettent aux différents processeurs de traiter différentes instructions sur un ensemble de données.
Cependant, la parallélisation des traitements est un processus difficile à réaliser. En effet, il dépend de l'architecture cible sur laquelle on va paralléliser les traitements. Il faut indiquer explicitement comment découper les traitements, comment distribuer les données, et il faut gérer la communication entre les différents processeurs.Aussi, est-il important de bénéficier d'outils qui permettent de faciliter cette opération, voire de l'automatiser.
Pour ce faire, voici quelques solutions proposées pour faciliter la parallélisation des traitements.

SYNDEX [Lavarenne 91b] (Synchronized Distributed Executive). Il s'agit d'un environnement logiciel graphique interactif de développement pour les applications temps réel de commande, de traitement du signal et d'image et qui supporte la méthode A^3, Adéquation Algorithmique Architecture.

Cette méthode a pour but de réduire le temps de développement et de spécification des applications temps réel. Elle permet d'aider à l'implémentation d'un algorithme sur une architecture donnée avec modification possible de l'architecture et de l'algorithme. L'adéquation correspond à la mise en correspondance efficace de l'algorithme, mis sous forme de graphe de flots de données et possédant un parallélisme potentiel, avec le graphe de l'architecture qui possède un parallélisme effectif. Le travail consiste à déterminer la transformation optimale permettant de respecter les contraintes temps réel en minimisant les ressources matérielles utilisées.
Pour réaliser ces transformations, SYNDEX utilise la spécification de l'algorithme et de l'architecture matérielle faite en langage SIGNAL (pour plus d'information sur ce langage voir l'annexe B. L'utilisation de ce langage formel synchrone va permettre des vérifications logiques sur la cohérence des événements des signaux en étudiant les horloges des signaux. L'utilisateur dispose d'une interface graphique avec laquelle il va spécifier l'algorithme et l'architecture sur laquelle il va être exécuté. L'implémentation de l'algorithme sur l'architecture consiste à distribuer les opérations sur les processeurs et à les ordonnancer. L'efficacité de cette implémentation est obtenue via des heuristiques d'optimisation concernant divers critères (tels que le temps de réponse ou la minimisation du nombre de processeurs utilisés).
Après cette distribution, un exécutif est généré automatiquement pour chaque processeur, grâce au noyau générique d'exécutifs de SYNDEX. Les exécutifs sont des programmes C et l'utilisateur dispose d'un fichier `make` qu'il va pouvoir modifier pour compiler et lier avec les fonctions d'interface qui sont appelées dans le programme SIGNAL et que l'utilisateur aura écrites.

Les squelettes fonctionnels. [Ginhac 98] proposent l'utilisation de *squelettes fonctionnels* prédéfinis qui correspondent à des structures génériques pour la parallélisation de traitements bas niveau et moyen niveau en traitement d'images. Les traitements sont réécrits en utilisant ces squelettes, ce qui rend transparent toute la gestion des communications et de la mémoire. Ces squelettes sont les suivants :

- SCM (Split, Compute and Merge). Ce squelette associe trois opérations qui effectue respectivement la partition des données d'entrée en sous-domaines (split), le calcul sur chacun des sous-domaines (compute) et la fusion des résultats intermédiaires (merge).
- DF (Data Farming). Ce squelette vise à assurer un équilibre automatique des la charge de calcul sur les processeurs de l'architecture cible. Il est basé sur le principe de la *ferme de processeurs* dans laquelle un serveur (farmer) envoie dynamiquement des données à traiter à un ensemble d'esclaves et accumule les résultats intermédiaires provenant des ces mêmes esclaves.
- TF (Task Farming). Ce squelette est une généralisation de DF. Chaque esclave peut récursivement générer plusieurs paquets de données à traiter.
- ITERMEM. Ce squelette est utilisé pour traiter explicitement les boucles sur un flot de données, c'est-à-dire quand le résultat d'un calcul à l'itération i est utilisé à l'itération $i + 1$.

De plus, l'environnement de programmation associé, SKIPPER [Serot 99], a été créé pour permettre le prototypage rapide d'algorithmes de vision sur des plateformes *MIMD*, utilisant ces squelettes fonctionnels. SKIPPER utilise de manière transparente l'environnement SYNDEX, qui va gérer la génération de l'exécutif en fonction de la topologie de la machine cible.

- La gestion des paramètres des traitements.
  On a vu que le temps de calcul des algorithmes bas niveau utilisés en vision dépend le plus souvent de la taille des données à traiter. Or, pour les algorithmes plus haut niveau, la taille des données peut être variable en fonction du résultat du bas niveau et elle évolue dynamiquement (la complexité n'est plus linéaire en fonction des données). Il est donc nécessaire de pouvoir paramétrer les algorithmes pour maîtriser la quantité de données produites en sortie, car elle conditionne le temps de calcul des traitements suivants.
  [Crowley 94] explique qu'un processus de vision s'exécutant continuellement doit pouvoir limiter ses calculs à un petit sous-ensemble de données obtenues à partir des capteurs visuels et il doit pouvoir adapter dynamiquement ces modes de calcul en fonction des événements de la scène observée ou des besoins de l'application.
  Cependant, la gestion de ce paramétrage peut s'avérer très complexe. En effet, si on considère la chaîne de traitements suivante : `lissage, recalage, extraction de coins, mise en correspondance avec l'image précédente`, le temps de calcul de la mise en correspondance va dépendre fortement du nombre de coins extraits qui dépend lui-même du lissage qui a été effectué.

3.2 Supervision des modules de traitements & IHM

Dans ce mémoire, nous étudions la supervision comme un processus qui requiert l'assistance d'un opérateur humain. La supervision qui nous intéresse s'apparente aux interactions à travers une interface homme-machine entre un utilisateur et une application.

De façon générale, une interface homme-machine permet à un utilisateur d'être à la fois **informé** en temps réel, c'est-à-dire avoir accès à des données concernant l'exécution de l'application, mais aussi de pouvoir **influer** sur le déroulement de l'application en cours.

Le type d'application qui nous intéresse (c'est-à-dire vidéo-surveillance) peut être utilisée conjointement à une interface graphique dont les caractéristiques sont les suivantes :

i. **Informer** l'utilisateur de ce qui se passe dans l'application.
   - Elle permet de visualiser une(des) image(s), les paramètres, les événements, des résultats de traitements (par exemple, le résultat de la détection et du suivi qui comprend l'affichage de la zone suivi sur l'image, l'affichage des valeurs caractéristiques de cette zone, etc.).
     Cette phase d'information peut être appelée de manière périodique si la visualisation des résultats le requiert.

   L'interface IHM en vision peut être la source d'un flot de données important (paramètres, données sur l'image). Mais ce flot est généralement de fréquence plus basse que le flot de données image provenant du capteur caméra.

ii. **Influer** sur l'application.
   - Elle fournit des données d'entrée pour les modules de traitements algorithmiques (points d'intérêt dans une image, zone cible à traiter, etc.).
   - Elle fournit des paramètres d'entrée aux traitements algorithmiques (paramètres de la caméra type zoom, focus, pan, tilt, seuil de lissage, taille de la fenêtre de corrélation, etc.).
   - Elle réagit de manière discrète par l'intermédiaire d'événements pour lancer, arrêter des traitements.

Cette IHM peut revêtir différentes formes en fonction de l'application, des besoins graphiques, etc. En effet, il peut s'agir d'une machine à retour d'efforts relié à un ordinateur (cas en imagerie médicale), d'une interface graphique avec visualisation d'images provenant d'une caméra (qui peut être pilotée à travers l'interface). Aussi la gestion de cette interface avec l'application doit être suffisamment flexible pour permettre, par exemple, de changer dynamiquement la configuration de l'IHM en fonction de l'application ou de la dynamique de la scène.

La partie qui gère les traitements temps réel doit être autonome vis-à-vis du processus de supervision, dans le sens où elle peut s'exécuter même si la supervision est indisponible.

3.3 Aspect contrôle et raisonnement

Si nous revenons aux applications qui nous intéressent, nous avons vu quels sont les types de traitement de vision qui peuvent être utilisés, comment on peut gérer les contraintes temporelles. Cependant, suivre une personne c'est bien mais interpréter et faire des raisonnements sur ce qu'il se passe c'est mieux.

Par conséquent, l'application doit comprendre une partie contrôle et raisonnement qui va permettre d'analyser ce qu'il se passe dans la scène et d'agir en conséquence.

Nous allons présenter dans la suite différentes approches qui permettent de gérer cet aspect raisonnement et contrôle par l'intermédiaire des techniques d'Intelligence Artificielle.

DAMN [Rosenblatt 97]. Distributed Architecture forMobile Navigation est une architecture réalisée par le CMU [Rosenblatt 97] et intégrée dans les véhicules NAVLAB pour faire du suivi de route, de la navigation en campagne, faire de la téléopération avec évitement d'obstacles.

Cette architecture 3.2 lance de manière concurrente (similaire à la *Subsumption Architecture* proposée par Brooks [Brooks 86]) différents modules, appelés *comportements*, qui se partagent le contrôle du robot mobile. Chacun de ces modules est responsable d'un aspect particulier du contrôle du véhicule ou réalise une fonction particulière. Ces modules vont envoyer un vote, concernant un ensemble possible d'actions sur le véhicule, à une entité particulière dénommée "arbitre". DAMN considère un `turn arbiter`, pour décider de combien on va tourner le volant, un `speed arbiter` pour décider de la vitesse du robot ou encore un `field of regard arbiter` pour décider où doit aller le robot.

A chaque comportement, on assigne un "poids" qui reflète sa priorité courante à contrôler le robot. Un gestionnaire de priorité peut être utilisé pour faire varier ces poids au cours de la mission. Les "arbitres" vont alors combiner les votes et les poids associés avec une

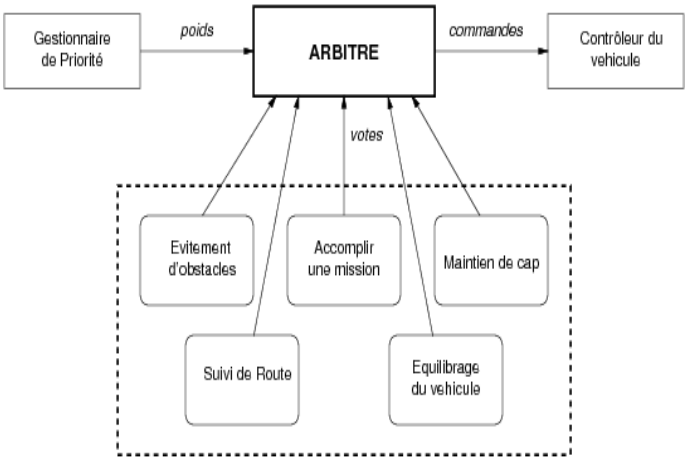


FIG. 3.2 – Structure générale de DAMN

stratégie dépendant de l'arbitre (une somme pondérée, convolution avec un masque gaussien, utilisation de logique floue, etc.) et c'est la commande dont le vote a la plus grande valeur qui est choisie pour piloter le robot.

Cette architecture a été validée expérimentalement par l'expérimentation menée par le CMU avec les véhicules NAVLAB.
Cependant, elle ne fournit pas d'outil pour assurer que la commande la plus appropriée à un instant donné sera bien exécutée par le robot. De plus, au niveau de la gestion de la sécurité du robot, rien de dit malgré la gestion de poids permettant de modifier les priorités des modules participant au choix de la commande, que le robot va bel et bien éviter un obstacle.

MESSIE [Giraudon 96]. MESSIE est une architecture générique d'interprétation de scènes, développée sous le générateur de systèmes experts SMECI (produit Ilog).

Cette architecture est hiérarchique et centralisée autour d'un tableau noir.
L'interprétation de scènes repose sur l'utilisation de plusieurs bases de connaissances ou *spécialistes* qui vont coopérer pour résoudre un problème d'interprétation.

Les spécialistes sont répertoriés selon 3 niveaux hiérarchiques du haut vers le bas niveau :

1. les spécialistes de niveau scène.
   Ces spécialistes servent à définir la stratégie globale de l'interprétation. En pratique, il n'existe qu'un seul spécialiste de ce haut niveau qui gère l'interprétation et les conflits de localisation entre les objets détectés.
   Le formalisme des ensembles flous et la théorie des possibilités sont utilisés ici pour gérer les incertitudes sur les objets détectés et décider de leur interprétation.
2. les spécialistes de niveau objet sémantique.
   Ces spécialistes servent à détecter des objets dans la scène. On a un spécialiste par type d'objet à détecter. Les objets sémantiques sont décrits, quant à eux, selon 4 points de vue : géométrie, radiométrie, contexte spatial et fonctionnalité.
3. les spécialistes de niveau traitement d'image.
   Ces spécialistes extraient les primitives images sur requêtes des spécialistes de ni-

veau supérieur. Ils connaissent le modèle des paramètres des algorithmes de traitement d'image, le modèle du capteur et quelles décisions prendre pour le choix des paramètres et la commande du capteur.
Les décisions prises par les spécialistes de niveau traitement d'image sont faites par le biais d'une mesure d'erreur entre le résultat de l'algorithme et le modèle des primitives attendues, pouvant être lui-même modifié par des contraintes (spatiales pour limiter le nombre de primitives extraites et sur le type des primitives tel que *trouver des petites régions*) provenant des niveaux supérieurs.

Les différents spécialistes vont déclarer sur le tableau noir leurs compétences sous forme d'objets de type *guichet* et les événements qui les intéressent.
Pour les spécialistes de bas niveau, ce sont les *guichets* qui sont chargés de faire le contrôle de la(es) tâche(s) à exécuter (ajustement de paramètres, calcul de la mesure d'erreur).

Le contrôle assuré par le tableau noir se fait :

- soit par requêtes : lorsqu'une tâche a besoin d'information, elle envoie une requête au tableau noir qui se charge de trouver le spécialiste le plus à même d'y répondre.
- soit par réception d'événements : le tableau noir, en fonction des déclarations faites au départ par les différents spécialistes, lance alors le spécialiste qui attend cet événement. Deux types d'événements peuvent être gérés : les événements internes provoqués par le raisonnement en cours dans le système, et les événements externes qui proviennent de l'environnement. Ce mécanisme permet de gérer en fait la réactivité du système et permet ainsi de définir son comportement. Cependant, contrairement à ce qui a été vu pour les langages synchrones, ce type de comportement sous MESSIE ne permet pas son analyse formelle.

Les capteurs (radar, infra-rouge, télémètre ou caméra) sont décrits par leur bande passante, leur type (actif, passif) et leur sensibilité aux aspects des matériaux et aux aspects géométriques des objets. (la réponse d'un radar est sensible à la forme des bâtiments, ou encore un infrarouge est sensible à la chaleur d'un objet). Le point de vue radiométrique d'un objet sémantique associé à la sensibilité du capteur permettent d'effectuer des raisonnements symboliques pour le choix du meilleur capteur en fonction de l'objet à détecter. De plus les capteurs commandables possèdent une liste de leurs paramètres de commande et de leurs valeurs (zoom, focus, inclinaison, etc.).

Ce système a été validé expérimentalement sur un système d'interprétation de scènes d'intérieur tridimensionnelles pour un robot mobile.

Pour cela, les données en entrée du système sont des segments 3D construits par stéréovision trinoculaire et les spécialistes utilisés sont les suivants :

1. les spécialistes robotiques :
   - spécialiste de but-robotique :(spécialiste de scène mais pour réaliser un but robotique) il a la charge de réaliser une mission sélectionnée par l'utilisateur en la traduisant en suite de buts de vision et d'actions robotiques. Deux types de missions ont été mis en oeuvre : la création d'une carte de l'environnement du robot et la localisation du robot sur la carte.
   - spécialiste d'action-robotique : il a la charge le déplacement du robot et de sa tourelle.
2. les spécialistes de vision bas-niveau :

- spécialiste de détection de contour ;
- spécialiste de mise en correspondance et de reconstruction ;
- spécialiste de groupement des segments pour la construction et la classification des facettes.

3. les spécialistes de vision d'objets sémantiques :
   - spécialiste de détection des tables ;
   - spécialiste de détection des bureaux ;
   - spécialiste de détection des chaises.

Ce système a été validé expérimentalement sur un système d'interprétation de scènes d'intérieur tridimensionnelles pour un robot mobile.
Cette approche prend en compte explicitement les capteurs et gère la réactivité avec l'environnement (à travers le raisonnement dirigé par les données avec le mécanisme des événements). Cependant, il ne permet pas de faire des vérifications au niveau de la validité logique du système ou encore au niveau temporel.

3.4 Intégration

3.4.1 Principe

D'après [Robert 97], l'intégration représente :

la coordination nécessaire au fonctionnement harmonieux, des activités de différents organes.

Si on transpose cette définition du domaine physiologique au niveau des systèmes informatiques, l'intégration peut être définie comme :

l'interface nécessaire au fonctionnement cohérent des activités des différents composants d'un système informatique.

Dans le cadre de notre travail, une intégration optimale s'occupe de coordonner les aspects supervision, raisonnement et traitements algorithmiques, ces derniers devant satisfaire aux contraintes temporelles imposées par l'application.

Dans ce qui suit, nous allons présenter différentes approches qui s'intéressent à l'intégration de tous ou partie de ces aspects, et étudier ce qu'elles proposent au niveau validation, sûreté de fonctionnement et méthodologie de programmation.

3.4.2 Méthodologies pour la vision par ordinateur

La perception active [Bajscy 88]. Le but de cette méthodologie est d'acquérir de manière "intelligente" les données nécessaires en vision pour résoudre un problème donné.
Et pour cela, Bajscy propose d'étudier la modélisation et les stratégies de contrôle de la perception visuelle, c'est-à-dire, la modélisation des capteurs (caméra, radar, laser, etc.), des objets, de l'environnement et de l'interaction entre eux.

Pour ce faire, elle propose une modélisation en deux parties :

- un modèle "local", qui caractérise au sein des modules de traitement les paramètres des capteurs et des algorithmes de traitement d'image (par exemple, taille d'un filtre ; la résolution spatiale) et la mesure d'erreur de leurs résultats. Ce modèle permet la prédiction des résultats du module de traitement en fonction des paramètres.
- un modèle "global", qui représente l'interaction entre les différents modules (comment on combine les résultats des différents modules entre eux). Cette interaction se fait par une boucle de retour qui permet aux modules de chercher les données sur demande quand elles deviennent nécessaires.
  Le modèle global intègre les paramètres globaux, ainsi que l'état initial et final du système.
  Ce modèle représente une sorte de contrôle "intelligent" du système de vision.

La stratégie de la perception active intègre des processus de raisonnement, de décision et de contrôle :

- il y a 3 étapes de contrôle mises en séquence au niveau de l'action à réaliser : à l'initialisation, pendant l'exécution, résultat final ;
- Au niveau raisonnement et décision, on va rechercher une séquence d'actions (mettant en oeuvre aussi bien les capteurs que les traitements au niveau des images extraites des capteurs) qui va permettre d'obtenir un maximum d'informations moyennant un coût minimal et atteindre le but fixé.
  Selon la provenance des informations (d'une connaissance "a priori" obtenue grâce à un modèle ou provenant des données acquises à travers les capteurs), on distingue 2 types de stratégie de contrôle : une guidée par les données (*bottom-up*) et l'autre guidée par une connaissance a priori (*top-down*). Dans les deux cas, le processus de décision fait appel aux techniques d'estimation avec une fonction de coût à minimiser, pour dans un cas (*bottom-up*), déterminer quel est le modèle correspondant aux données provenant des capteurs et traitées (par exemple : ensemble de régions, de lignes, de points 3D), ou dans l'autre (*top-down)*, rechercher des données provenant des capteurs qui vérifient la validité du modèle proposé.

Deux validations expérimentales de cette approche ont été réalisées sur un système à 11 degrés de liberté muni de deux caméras, le système *Agile Camera* et concerne le cas d'une stratégie de contrôle guidé par les données :

1. Construction d'une carte de profondeur en combinant les informations provenant des modules focus et stéréo. L'accent est mis sur le contrôle de la caméra nécessaire à l'initialisation au niveau du focus, du zoom, de la vergence, pour acquérir les données nécessaires à la construction de la carte de profondeur.
2. Segmentation d'une image 2D réalisée par la coopération d'un module de détection de contour et un module de formation de régions. Chacun de ces modules a été modélisé selon le modèle local, c'est-à-dire en explicitant chacun de leurs paramètres et les fonctions d'erreurs associées. Le modèle global représente le flot de données entre les modules, l'interaction entre les modules (réalisée par ajustement des paramètres) et les données globales (taille et nombre de régions approximatif).

Cette approche prend en compte du capteur dans le processus de perception (phase d'acquisition des images dans le processus de perception visuelle) La modélisation proposée prend en compte les paramètres des modules de traitement et des capteurs. De plus, la dy-

namicité du comportement du système est assurée grâce à la boucle de retour. Cependant, aucun formalisme clairement spécifié n'est associé à cette modélisation. De plus, au niveau de la vérification, l'approche ne fournit pas de mécanismes explicites au niveau formel que ce soit pour le contrôle ou pour la partie traitement algorithmique.

La vision intentionnelle et qualitative [Aloimonos 90] : Cette méthodologie propose de construire des systèmes de vision robustes réalisant des tâches relativement complexes et utilisables en industrie en extrayant uniquement les informations pertinentes en fonction de la tâche à réaliser (résoudre les problèmes de vision en se focalisant sur le BUT du système que l'on souhaite construire).

La vision intentionnelle permet de décomposer un problème initial en plusieurs sous-problèmes, et définir les modules de traitement associés à chacune des sous-tâches, pour concevoir le superviseur qui activera ou pas telle ou telle tâche.

De plus, Aloimonos propose la notion de vision qualitative qui repose sur le fait que beaucoup de mesures dans un système de vision résultent en une mesure discrète (oui/non, etc.) et ne nécessitent pas une grande précision. Ce qui va permettre des filtrages sur les actions à lancer en fonctions des réponses discrètes.

La validation de ces concepts a été faite à travers le robot *Medusa* (muni d'une caméra et d'un bras manipulateur). Ce robot devait détecter si un objet se déplaçait autour de lui, le suivre et l'intercepter si il se rapprochait du robot.
Pour cela, la donnée en entrée du système est une série d'images, et les modules utilisés sont les suivants :

- calcul de la composante normale du flot optique en chaque point ; en sortie de ce module on a une information sur le mouvement détecté et l'emplacement de l'objet en mouvement dans image ;
- déterminer si l'objet se rapproche ;
- suivi de l'objet en déplacement en le gardant au centre de l'image ;
- prédiction la région d'impact dans l'image et prédire si l'objet va heurter le robot ;
- commande du robot pour intercepter l'objet en mouvement.

Cette approche a été validée expérimentalement dans le cas d'un système de navigation de robot. Elle permet l'adéquation des modules de traitement avec la tâche à effectuer. De plus, la vision qualitative permet de palier aux imprécisions des mesures extraites des images suivant le raisonnement que l'on adopte (répondre oui/non). Cependant, l'interaction des modules qui composent les tâches complexes de vision est statique : il n'y a pas de gestion dynamique entre les modules en fonction d'un but à atteindre ou de l'état du système.

La "Tâche Vision" [Ikeuchi 96]. Cette méthodologie propose de définir une approche *orientée tâche* qui permettra au système de vision en fonction du but à accomplir et de l'environnement dans lequel le but doit être réalisé, de modifier son architecture de manière à sélectionner les composants qui satisferont au mieux le but à réaliser.

Cette approche se différencie de l'approche *généraliste* où les systèmes de vision sont construits dans le but de résoudre toutes les tâches de vision en utilisant une unique architecture. Les composants de l'application de vision sont alors sélectionnés une fois pour toute et sont exécutés dans un ordre fixé.

[Ikeuchi 96] définit un *paradigme orienté tâche* pour lequel on considère :

1. la définition de la spécification de la tâche à accomplir ;
2. la définition des besoins fonctionnels de la représentation modélisant la tâche ;
3. la définition des caractéristiques appropriées pour extraire les représentations définies au point précédent ;
4. la définition des méthodes de segmentation appropriées pour extraire les caractéristiques et les représentations déterminées précédemment ;
5. la caractérisation des capteurs d'image et des drivers appropriés pour obtenir les méthodes de segmentation et les caractéristiques précédemment définies.

L'élément-clé est *l'ordre logique* dans lequel les composants de vision vont être sélectionnés et construits. Cette sélection des composants est basée sur les contraintes imposées par le type de tâche vision à élaborer.

Ikeuchi illustre son paradigme avec les exemples suivants :

– la saisie par un robot muni d'une main articulée de pierres dans un site lunaire.
  Là, on n'a pas besoin de faire des calculs précis pour saisir un objet. Donc on travaillera sur des valeurs approchées pour modéliser l'objet à saisir, et on emploiera par exemple une modélisation superquadratique (correspondance entre une ligne ou une surface quadratique et un objet).
– la saisie d'objets dans une chaîne de montage.
  Ici, on doit établir le modèle exact de l'objet à saisir. Donc on pourra utiliser un modèle géométrique polyhédrique.

Cette approche fournit une définition de la "tâche vision", c'est-à-dire une entité élémentaire avec laquelle une application va être structurée. Cependant, cette définition n'explicite pas formellement le contrôle associée à la tâche, et ne propose pas d'environnement ou d'outils associés.

3.4.3 Architectures/ Environnements de programmation

Dans ce paragraphe, nous allons étudier un sous-ensemble représentatif d'environnements de programmation d'applications de vision. Ces environnements fournissent des outils dédiés aux différentes parties de la mise en œuvre des applications ainsi qu'une architecture particulière qui intègre les différents composants de l'application.

OCAPI [Clément 93]. OCAPI est un système à base de connaissance dédié au pilotage de programmes conçu et développé comme un générateur de systèmes expert. Le pilotage de programmes [den Elst 96] consiste à sélectionner et adapter les modules par rapport à un but donné et aux ressources disponibles.

OCAPI comprend une librairie de programmes exécutables, une base de connaissance contenant des informations sur ces programmes et la façon dont ils doivent être appliqués pour résoudre un problème particulier ; ainsi qu'un moteur de supervision qui va gérer l'ensemble.

Le moteur de supervision est composé de quatre modules :

– un planificateur qui va déterminer un plan en fonction du but à atteindre. Ce plan est constitué d'un ensemble de procédures.
– un module d'exécution du plan établi par le planificateur. Il va lancer les procédures implémentant le plan, initialiser des paramètres si nécessaires en utilisant la base de connaissance.
– un module d'évaluation des résultats obtenus après exécution du plan. Il va utiliser les résultats attendus contenues dans la base de connaissance pour comparer les résultats obtenus.
– un module de corrections dans le cas où les résultats obtenus ne sont pas corrects. Soit le plan est modifié, soit on réexécute la procédure fautive avec un paramétrage différent.

La base de connaissance contient la modélisation des entités suivantes :

– le but :
  Il décrit les propriétés à vérifier pour atteindre un objectif donné. Pour décider entre plusieurs méthodes répondant à l'objectif, le but est caractérisé par des règles de choix décidant de la meilleure action à lancer pour répondre à l'objectif et par des règles d'évaluation des résultats attendus indépendants de l'action à lancer.
– l'opérateur :
  Il décrit le ou l'ensemble des modules qui peuvent être réalisées pour répondre à un but. Il se caractérise par sa fonctionnalité, les arguments d'entrée et de sortie (par exemple : une séquence d'images), les paramètres (par exemple : un seuil de lissage), les règles d'initialisation et de réajustement associées à ces paramètres, les préconditions (par exemple : pas de mouvement dans l'image), les effets ou postconditions (par exemple : la donnée de sortie contient la localisation spatiale d'un objet), les règles d'évaluation les arguments de sortie et les règles de reprise sur erreur dans le cas où les arguments de sortie ne soient pas corrects.
  Le corps des modules est écrit sous forme d'un script shell, par exemple :
  ```
  calseuil -s1 fact1 -s2 fact2 -s3 fact3 profil-moyen.get-filename 1>inter
  ```
  où `facti` est un paramètre compris entre $[0.0, 1.0]$,
  `profil-moyen.get-filename` représente la données d'entrée,
  `inter` la donnée de sortie et
  `calseuil` le module de l'opérateur de calcul de seuil.

– la requête :
  Elle définit quel but doit être atteint, avec quelles données d'entrée et avec quelle qualité de résultats.
– le contexte :
  Il décrit les valeurs possibles des caractéristiques des données d'entrée. (par exemple : données pas, peu ou fortement bruitées, mouvement dans l'image, importance des coins dans l'image).
  Cette information sera utilisée par les règles de choix pour décider quel opérateur va être choisi.
– les données :
  On distingue les données de type *argument* (d'un opérateur ou d'un but) des données de type *paramètre*. Ces données sont décrites de manière structurées : une image est

représenté par une structure qui contient des informations sur les conditions d'acquisition sur les objets que l'on peut y observer, sur ses paramètres et sur la façon dont on peut accéder au données contenues dans l'image.

Ce système a été validé sur des applications en traitement d'images médicales [Crubézy 97], pour la classification de galaxies [Vincent 97], et pour de l'interprétation dynamique de scènes à partir de séquence d'images [Bremond 97].

L'aspect vérification est envisagé par [[Marcos 95],[Marcos 97]] au niveau de la base de connaissance : le but est de vérifier la cohérence de la base de connaissance et sa complétude, mais aussi de vérifier l'adéquation de la connaissance avec son application faite par l'utilisateur.

A niveau des règles de connaissance :

- vérification syntaxique. On vérifie que chaque objet référencé existe, que les identificateurs sont liés à un objet ou à une variable.
- vérification de la règle de connaissance à la fonction du composant associé à la règle.
- vérification de la sémantique des règles.
- vérification de la consistance des règles : pas de règles redondantes, de règles contradictoires au niveau résultats et au niveau pré-requis.
- vérification de la complétude de la base de connaissance : détecter s'il manque des attributs non référencés.

[Marcos 95] présente un outil de "parsing" (à base de YACC) qui regarde si syntaxiquement les règles sont bien construites et si les objets référencés existent bel et bien.
Cet outil a déjà été utilisé pour un base de connaissance de détection d'obstacles dans des scènes urbaines et sur des routes [Bremond 97].

Cette approche gère explicitement la fonctionnalité des opérateurs. Elle permet une adaptation dynamique du paramétrage et des modules liée à un processus de prise de décision. Ce mécanisme gère explicitement l'initialisation des paramètres des opérateurs (*initialisation criteria*), leur ajustement (*adjustment criteria*) et l'évaluation des résultats des opérateurs (*evaluation criteria*). Il permet de plus de traiter dynamiquement l'exécution défaillante d'un opérateur (*assessment et repair criteria*).
La vérification de la validité de la base de connaissance est prise en compte mais pour l'instant cette vérification se cantonne à une analyse syntaxique sur l'ensemble des règles d'une base de connaissance.

Cette approche ne permet pas la prise en compte explicite des contraintes temps réel liées aux séquences d'images (cadence, fréquence d'acquistion).

RAVI [Zoppis 97, Lux 97]. RAVI (pour Robotique, Apprentissage, Vision et Intégration) est une plateforme multi-langage qui a pour but d'intégrer les différentes techniques suivantes : robotique, vision et apprentissage.

Cette plateforme est composé d'une machine virtuelle au-dessus de laquelle sont implantés les différents langages de programmation SCHEME, PROLOG, et un système de règles particulier RISP (Ravi Integrated Product System), et elle permet de charger dynamiquement des bibliothèques de modules algorithmiques C ou C++. Ces modules concernent le traitement d'images, l'extraction de segment, l'utilisation de X-WINDOW ou le contrôle d'un robot mobile.

Le module d'utilisation de X-WINDOW permet de créer des interfaces graphiques avec l'outil TK (avec fenêtres, menus, boutons, saisie de texte ou affichage graphique) qui sont capables d'appeler des fonctions de la machine virtuelle de RAVI sous forme de *callback*. La plateforme facilite la mise en relation des programmes relevant de différents domaines tels que la robotique ou la vision, pour les intégrer dans une même application.

La plateforme génère automatiquement l'interface entre les bibliothèques C, C++ et les langages tels que SCHEME, PROLOG. Ceci permet d'utiliser conjointement les algorithmes numériques et les techniques de raisonnement à base d'Intelligence Artificielle.
La plateforme fournit également des interprètes au-dessus de la machine virtuelle. Ils permettent un accès interactif aux fonctionnalités des bibliothèques ce qui facilite les expérimentations.

Le contrôle des applications qui sont construites par RAVI se fait sous la forme de *graphe de contrôle reconfigurable dynamiquement*. Ces graphes permettent de représenter explicitement le séquencement, et la parallélisation des modules, la détection de dysfonctionnement.

Ces graphes sont modélisés sous la forme de modules particuliers. reliés par des canaux de communication.

VAP (Vision as Process) [Crowley 94]. VAP représente un projet européen regroupant différentes équipes de recherche dont le but a été de construire un *système intégré de vision active* fonctionnant en continu et en temps réel.
Ce projet n'envisage la vision que comme un processus qui doit s'intégrer dans une chaîne de traitement plus grande. La vision n'est ici pas considérée comme une fin en soi, mais pour fournir les informations pertinentes concernant l'environnement pour répondre à une des prérogatives de l'application.

Le système proposé a été décomposé en six modules s'inscrivant chacun sur un niveau de représentation :

- l'unité de contrôle de la caméra ;
- la description 2D ;
- la description tridimensionnelle ;
- l'interprétation symbolique de la scène observée ;
- le superviseur responsable du contrôle du système ;
- et enfin un module dédié à la spécification du contrôle du système.

L'architecture du système est composée d'une collection de modules indépendants communiquant par le biais de :

- canaux dédiés pour l'échange de données ;
- messages pour les informations de contrôle.

Chaque module est implanté comme un processus UNIX indépendant lié aux autres par un *squelette*. Ce squelette est nommé SAVA (Squelette d'Application pour la Vision Active). Il définit l'interface commune pour tous les modules.

Les modules gérés par le système SAVA sont structurés de manière standard comme illustré à la figure 3.3. En fonction des besoins du nouveau module à créer, l'utilisateur va spécialiser cette structure standard.

Le noyau du système est composé d'un interpréteur de règles basé sur le système CLIPS

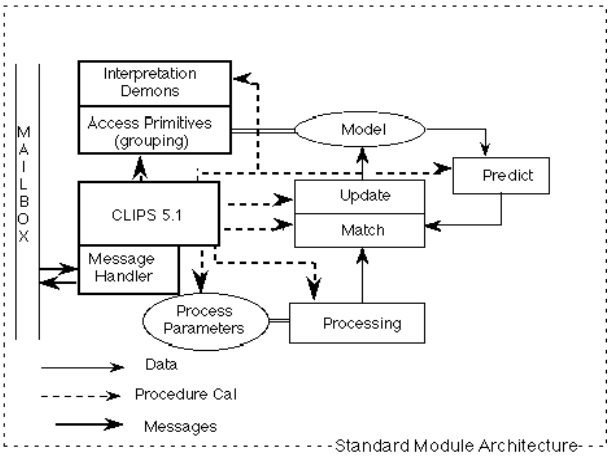


FIG. 3.3 – Structure standard du module manipulé par SAVA.

5.1. Cet interpréteur gère l'ordonnancement des modules, la sélection des paramètres, le protocole de communication et la possibilité de définir des processus "daemon" qui réagissent aux événements.

Cette approche ne fournit pas de formalisme bien défini pour construire une application, à part le structure standard de module. Elle ne permet pas non plus, de vérification sur la composition de l'application basée sur le système CLIPS.

L'environnement ORCCAD/MAESTRO [Team 98, Turro 99] .

Le système ORCCAD est un environnement conçu pour spécifier, valider avec des méthodes formelles, simuler et implémenter des applications robotiques.
La création d'un tel système a été motivé par la difficulté de réaliser des tâches impliquées dans un processus robotique qui utilise une architecture de contrôle distribué, nécessitant de bonnes performances au niveau temps réel.

Dans ce cadre, une application est perçue comme la composition structurée et hiérarchiques en *Tâches Robot* (niveau fonctionnel) et en *Procédures Robot* (niveau applicatif).
Les TÂCHES ROBOT sont les actions robotiques élémentaires qui combinent des aspects comportementaux mais surtout des aspects liés à la dynamique du système qu'elles contrôlent. Les PROCÉDURES ROBOT servent à structurer de manière **logique** plusieurs TÂCHES ROBOT dans le but de réaliser un objectif global, les PROCÉDURES ROBOT adressent donc principalement les aspects réactifs du système.

Le langage MAESTRO est un langage dédié à la programmation de mission robotique. Il permet de manipuler à travers une programmation de style impératif l'arrangement logique de TÂCHES ROBOT spécifiés avec ORCCAD pour définir une application.

Cet environnement ORCCAD/MAESTRO gère le contrôle au niveau des TÂCHES ROBOT et des PROCÉDURES ROBOT de manière uniforme et suivant le même formalisme, à savoir celui des systèmes réactifs mis en œuvre en langage ESTEREL. Ceci permet donc des vérifications formelles sur la partie du contrôle logique des applications spécifiées avec ORCCAD.

Cet environnement propose un environnement de programmation qui permet gra-

phiquement de spécifier les différents composants d'une application (TÂCHES ROBOT,PROCÉDURES ROBOT) de valider la partie contrôle logique, et de générer le code exécutable temps réel correspondant.
En revanche, cet environnement ne permet pas explicitement de gérer des processus de type décisionnel.

3.5 Conclusion

L'inconvénient de beaucoup de ces travaux est le manque de méthodologie précise qui facilite la conception des applications et qui permette de valider de manière formelle le respect des contraintes liées à la sûreté de fonctionnement, la robustesse et le respect des contraintes temporelles.

En conséquence, nous avons choisi de traiter cette carence méthodologique en proposant une approche orientée tâche, ce qui écarte les approches fondées sur les techniques de l'Intelligence Artificielle.
Nous avons décidé de nous intéresser à la formalisation de la partie dynamique d'une application de vision (la partie qui gère les traitements temps réel), et de fournir une méthodologie de programmation associée.

Chapitre 4

Formalisation de la partie traitements "Temps Réel"

Nous allons nous intéresser aux systèmes de vision définis comme suit :

> un système qui fonctionne de manière continue avec son environnement (scène observée) et qui dialogue avec lui à travers des capteurs visuels et/ou une interface utilisateur. Un tel système doit exécuter des actions de perception dédiées à l'application pour laquelle il a été conçu. De plus, ce système doit délivrer les résultats dans un délai fixe.

Pour ce type de système, nous choisissons de le mettre en œuvre suivant un schéma **continu/discret** :

> une partie "continue" gére un flot de données continu, transformé par les différents traitements de calcul mis en œuvre par le système de vision depuis l'acquisition de l'image par un capteur visuel jusqu'à l'obtention de données extraites des images filtrées.
> ET
> une partie "discrète" qui conditionne l'évolution du flot de données continu : avant le démarrage des phases de calcul pour l'initialiser ; signaler les dysfonctionnements, indiquer qu'un objectif a été atteint ou encore adapter le paramétrage des calculs.

Dans la suite de ce chapitre, nous allons analyser en détail les trois tâches : élaboration de la carte de la scène, calibration et suivi de cible et qui correspondent au traitements de vision temps réel mis en œuvre dans l'application de la "mamy-sitter" présentée au chapitre 2. Cette analyse est faite en considérant à chaque fois la partie continue et la partie discrète. Cette étude nous permet ensuite de caractériser plus finement ces tâches et nous mener à la définition de l'entité de vision élémentaire de programmation : la TÂCHE VISION, et à la proposition de la méthodologie de programmation.

4.1 Analyse des tâches

Dans cette partie, nous allons analyser les différentes compétences définies au paragraphe 2.3. Pour chacune d'elles, nous allons étudier les algorithmes spécifiques à la vision nécessaires pour mettre en œuvre la compétence donnée. Nous allons aussi présenter les caractéristiques du flot de données, des capteurs visuels, des événements, des paramètres et des contraintes temporelles qui interviennent dans cette mise en œuvre.

4.1.1 Élaboration d'une carte de la scène observée

4.1.1.1 Principe algorithmique de la tâche

But : détecter et localiser les objets de la scène observée avant d'analyser les objets en mouvement : localiser les objets fixes mais d'intensité variable (par exemple, les pales d'un ventilateur, des scintillements, les images d'un poste de télévision, etc.) et élaborer une mosaïque représentant ces éléments.

Comment : La mosaïque est une image composée de cellules qui correspondent chacune à une orientation donnée de la direction du champs de vision de la caméra (voir l'annexe A.2). Pour élaborer cette carte, voici les différentes étapes à suivre :

1. Prendre de manière itérative la différence des images à l'instant t et à l'instant $t-1$ et moyenner la valeur absolue de la disparité résiduelle obtenue.
2. Si la disparité est non négligeable, cela est dû soit à des objets en mouvement soit à des scintillements. Dans le cas d'objets qui bougent, la localisation de la disparité varie contrairement à celle des scintillements qui reste fixe. Par conséquent, la valeur moyenne de la disparité non négligeable est représentative uniquement des objets fixes "parasites".
3. Dès qu'une région est détectée, elle est ajoutée à la cellule correspondante de la mosaïque.
4. Les positions en "pan" et "tilt" de la caméra sont modifiées pour pouvoir explorer la cellule voisine.

Cette action est paramétrée par les angles P (pour le pan) et T (pour le tilt) de la tourelle, la valeur du zoom z, le coefficient de lissage w et le nombre d'itérations nécessaires pour calculer la disparité N :

– pour couvrir tout le champ visuel, différentes valeurs de P et T vont être considérées. Étant donné que c'est autour de la zone fovéale, c'est-à-dire, $\pm 10deg$ autour de l'axe optique de la caméra, que l'on est le plus précis en fonction du modèle de la caméra envisagé, on peut facilement déterminer le découpage de l'orientation de la tourelle pour couvrir tout le champs visuel. Pour chaque valeur de P et de T, les valeurs des angles horizontaux $H = P \pm 10deg$ et verticaux $V = T \pm 10deg$ sont obtenus en moyennant la disparité non négligeable.

– le nombre N de valeurs nécessaires pour calculer la valeur moyenne est directement positionné par l'utilisateur en fonction du temps dont il dispose pour ce calcul. Plus précisément, la structure de l'algorithmique montre que :

$Temps \simeq k\,N + h$ avec k et h qui peuvent être déterminés à partir de quelques tours d'exécutions.

– la *valeur du zoom* z doit être fixée à une valeur minimale. Si la caméra zoome, le changement de la longueur de la focale entraînera une variation de la localisation du centre optique de la caméra et par conséquent, les rotations de la tourelle n'entraîneront pas de rotations pures de la caméra.
– le *coefficient de lissage* w est défini en fonction de la taille de la cellule de la mosaïque (voir à l'annexe A.2, équation (A.3)). Plus précisément, on utilise un filtre récursif [Deriche 87a] pour lequel il a été démontré [Viéville 94b] que le coefficient de lissage peut s'exprimer comme une taille en pixel de fenêtre alors que toutes les autres opérations différentielles peuvent être implantées comme de simples différences en utilisant l'image lissée. Ceci permet de relier le coefficient de lissage à une zone de l'image : on prend simplement $w = \sqrt{S}$.

L'action "élaboration d'une carte de la scène" est schématisée par le schéma de la figure 4.1.

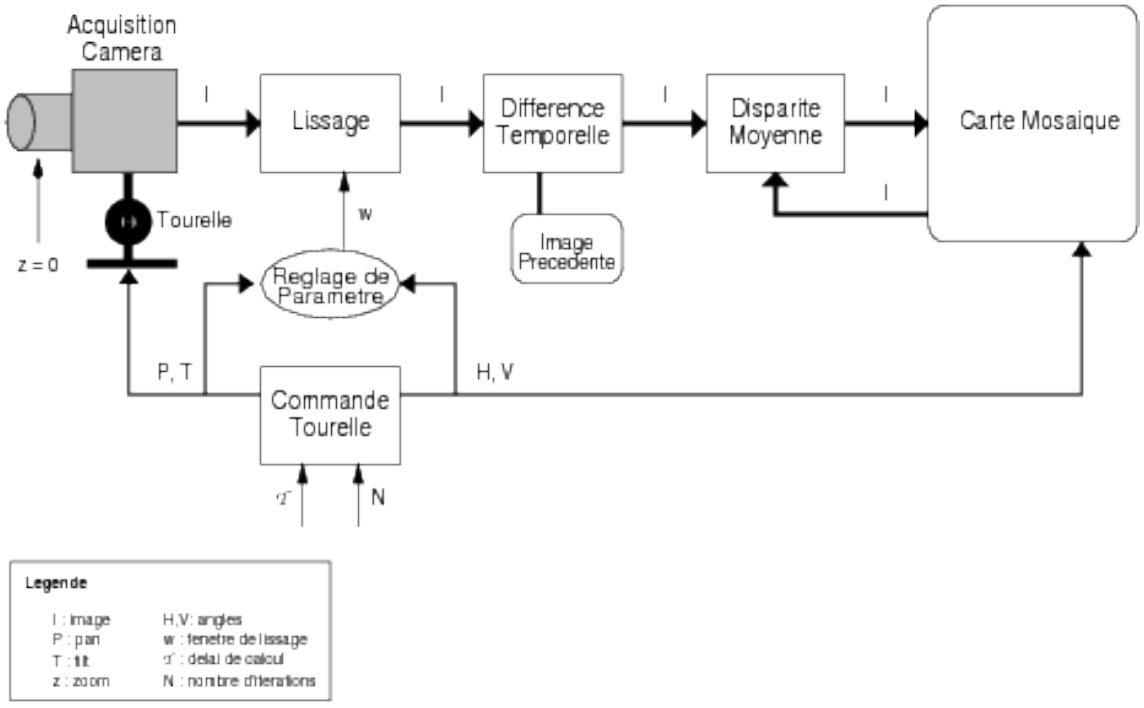


FIG. 4.1 – La séquence d'actions pour l'élaboration de la carte de la scène.

4.1.1.2 Contraintes de mise en œuvre

Maintenant que nous avons présenté l'algorithmique de la tâche, nous allons étudier les contraintes de mises en œuvre.

Flot de données. Le flot de données manipulé ici correspond à des images qui sont successivement transformées par les différents traitements définis dans la chaîne de vision. L'origine du flot provient du capteur visuel, c'est-à-dire la caméra, et il est transformé en une image mosaïque.
La transformation de ce flot de données est réalisée par une suite de traitements périodiques, chaque traitement est géré à l'intérieur d'un *module*. Du point de vue de la synchronisation, cela signifie que l'entrée d'un module est disponible quand le module précédent dans la chaîne a produit sa donnée de sortie. De plus, on peut régler la justesse de cette transformation par l'intermédiaire de paramètres. Ces paramètres proviennent soit du traitement (par exemple, le coefficient de lissage w), soit d'un processus extérieur (par exemple, interaction avec un utilisateur pour la valeur N).

Cette tâche introduit un premier type de transformation de flot de données : la transformation `im2im`, c'est-à-dire, image → image, **depuis** une image acquise par un capteur visuel **vers** une image de type mosaïque.
Les modules `im2im` nécessitent généralement beaucoup de calculs mais sur des types de données simples et avec des opérateurs bien définis. Généralement, la complexité de ces modules peut être considérée comme étant linéaire avec les données transformées.

Paramètres. Les paramètres peuvent être vus comme des régulateurs du processus de transformations du flot de données. Ils peuvent agir sur la qualité du résultat de la transformation : par exemple, si le coefficient de lissage est trop bas, on prend en compte en tant qu'information nominale un taux important de bruit et par conséquent, ceci produit la détection de faux objets. D'un autre côté, si le coefficient est trop élevé, on perd en précision de détection.

Cette action introduit deux types de paramètres :
- des paramètres fixés une fois pour toute en prémisse de l'action. Par exemple, la valeur du zoom z ou encore les valeurs successives des positions en pan et tilt.
- des paramètres générés par une fonction spécifiques. C'est le cas du coefficient w, qui est obtenu à partir de l'équation $w = \sqrt{S}$, où S est la surface rétinienne (pour plus de détails, consulter à l'annexe A, l'équation (A.3)).

Contrairement au flot de données qui est transformé cycliquement, les paramètres ne sont pas soumis à la même cadence. Par conséquent, les paramètres représentent des données à différencier explicitement du flot de données.

Contraintes temporelles. La tâche est réalisée de manière cyclique mais la période entre l'acquisition du flot de données source et la fin de la transformation n'est pas critique.
Cependant, la complexité algorithmique des modules $image \rightarrow image$ dépend linéairement de la quantité de données à traiter en entrée de chaque module algorithmique composant l'action.

Donc, pour une taille d'image donnée, le temps de calcul de la chaîne de traitements peut être déterminé. Cette remarque peut être très utile lorsque l'action doit être réalisée suivant un délai fixé. Mais ceci n'est pas vrai pour tous les types d'algorithmes, par exemple, ceux à base de recuit simulé. Dans ce dernier cas, on pourra fixer à l'avance le nombre d'itérations

mais on ne pourra plus garantir la justesse du résultat.

Capteurs. L'élaboration de la carte de la scène sous-entend l'utilisation d'un capteur visuel qui fournit une image de l'environnement observé. Ce capteur peut être vu comme une interface entre le processus de transformation du flot de données et l'environnement.
De manière cyclique, le capteur acquiert une image et modifie les positions en pan et tilt. La longueur du cycle dépend du temps pris par le processus de transformation du flot de données.

Étant donné qu'il est important de connaître l'état du capteur (en train de se déplacer), ainsi que les positions courantes de P et de T (P et T étant modifiées régulièrement), le capteur visuel doit être contrôlé pendant tout processus "oculomoteur". Dans le cas contraire, sans information concernant l'état du capteur, l'application est inutilisable.

Ce besoin d'assurer le contrôle des capteurs n'est pas nouveau : [Aloimonos 87] et [Bajscy 88] ont été les premiers à le mettre en exergue. Ils ont prouvé que les capteurs doivent faire partie du processus de vision pour permettre de résoudre efficacement les problèmes de vision "active".

Événements de contrôle. [Bajscy 88] décompose une action de vision suivant les phases suivantes :

1. *L'initialisation* : dans cette phase, on fixe le nombre de cellules de la mosaïque, la valeur du zoom, l'intervalle de valeurs du pan et du tilt, et l'utilisateur fournit une valeur pour N.
2. *L'exécution nominale* : cette phase correspond à l'acquisition, la transformation du flot de données, et au changement prédéfini de pan et du tilt.
3. *La fin* : à ce stade, toutes les cellules ont été explorées par le processus de vision cyclique et la mosaïque résultante est complètement construite.

Pour passer d'une phase à une autre, on considére des transitions de phases qui sont enclenchées par l'occurrence de conditions logiques particulières (ou des événements discrets).

- Conditions internes liées :
  - au capteur : "la caméra est-elle prête à commencer l'acquisition ?", "caméra KO", etc.
  - à l'initialisation de tous les modules algorithmiques : "initialisation achevée ?".
  - à la transformation algorithmique : "la qualité du lissage est-elle OK ?"
  - à la fin de l'action :"toutes les cellules ont-elles été explorées ?".
- Conditions externes liées :
  - à l'interface avec l'utilisateur :"démarrer action", "arrêter l'action".
  - à l'interface avec d'autres actions pouvant intervenir dans une application plus complexe : comment ordonner les différentes tâches, que faire dans le cas où une tâche s'exécute mal, qu'est-ce que cela implique pour les autres tâches, etc.

Ces phases et les conditions de transition de phase associées rythment l'évolution au cours du temps de la tâche de vision. On peut les comparer respectivement à un ensemble d'états et à la liste des événements qui permet de transiter d'un état à un autre.

En fait, la séquence d'états et des événements associés définit le comportement logique de la tâche de vision. Via ce comportement logique, le contrôle d'une tâche de vision est facilité et respecté. Et c'est aussi valable à un plus haut niveau, étant donné qu'une application est composée de plusieurs tâches de vision qui peuvent communiquer ou se synchroniser entre elles. Ceci souligne l'importance de la modélisation du comportement logique au niveau de la tache de vision.

4.1.2 Auto-calibration de la caméra par déplacements prédéfinis

4.1.2.1 Principe algorithmique de la tâche

But : calculer les paramètres de calibration intrinsèques : (u_0, v_0, f), propre au capteur visuel.

Comment : ces coefficients de calibration peuvent être évalués efficacement soit en considérant une calibration pure [Hartley 97] ou par rotation d'axe fixe [Viéville 94a]. Nous choisissons la deuxième solution.

Pour déterminer (u_0, v_0, f), nous utilisons les équations présentées dans A.3. Ceci implique de réaliser plusieurs mesures en des points caractéristiques de l'image en fonction de l'angle de rotation P de la caméra afin de résoudre le système linéaire défini par l'équation (A.4). Par conséquent, ce mécanisme nécessite de détecter des "coins" [1] dans l'image, comme proposé par [Smith 97]. De plus, il faut suivre et mettre en correspondance la position de ces coins dans la séquence d'image, comme présenté par [Brown 93], or étant donné qu'il faut gérer la disparité entre deux pour maintenir une petite différence entre chaque image consécutive, le processus de suivi devient évident.

Pour calculer finement les valeurs de (u_0, v_0, f), nous allons ajouter une boucle d'estimation qui fournit les coefficients de calibration. Pour ce faire, on a choisi de minimiser le critère suivant (exprimé pour f, mais également applicable pour u_0, v_0) :

$$\mathcal{L}_f = \sum_{i=1}^{N} e^{-\left(\frac{\epsilon_i}{s}\right)^2} \, [f_i - f]^2$$

où f_i et ϵ_i (erreurs de mesure) sont les mesures obtenues pour chaque point suivi, F est l'estimée de la focale et V_f est l'estimée de sa variance.

Le poids $\lambda = e^{-(\frac{\epsilon_i}{s})^2}$ doit être choisi de façon à être majoré mais maximal pour $\epsilon_i = 0$ (mesure "parfaite") et à décroître rapidement avec l'accroissement de l'erreur. Le choix d'une fonction exponentielle pour implanter cette propriété est arbitraire. Un paramètre s, donné en pixel, permet de paramétrer ce procédé.

Comme c'est un simple critère au moindre carré, l'équation précédente s'exprime comme :

$$f = \left[\sum_{i=1}^{N} e^{-\left(\frac{\epsilon_i}{s}\right)^2} f_i\right] / \left[\sum_{i=1}^{N} e^{-\left(\frac{\epsilon_i}{s}\right)^2}\right] \quad \text{avec} \quad V_f = \sum_{i=1}^{N} (f_i - f)^2/(N-1) \qquad (4.1)$$

[1] Ici, les coins sont des points caractérisés par une localisation bien définie dans l'image.

La qualité des estimations est quantifiée à travers le terme V_f.

Donc, voici les étapes à suivre pour déterminer les paramètres intrinsèques de calibration :

1. faire une rotation ;
2. calculer les déplacements rétiniens des points caractéristiques, c'est-à-dire, les coins ;
3. calculer les paramètres intrinsèques à partir des déplacements déterminés précédemment et en résolvant les équations (A.4) et (A.5) ;
4. Moyenner pour tous les points suivis, en résolvant les équations (voir le paragraphe A.1, équation (4.1)) ;
5. Si la qualité des valeurs n'est pas suffisante, reboucler.

Cette action est paramétrée par :
– l'amplitude de rotation P_{max}.
– le nombre de pas N, qui peut être choisi facilement pour fournir des disparités de l'ordre du pixel entre chaque image consécutive afin d'assurer une mise en correspondance des coins suivant une configuration optimale. On obtient : $N \simeq P_{max} \times \tilde{f}$ où $\tilde{f}$ est une estimation a priori de la focale.
– la valeur du zoom Z est échantillonnée suivant un intervalle de données prédéfinis $[position_min...position_max]$ pour estimer (u_0, v_0, f) pour différentes configurations de zoom.
– l'angle du tilt est maintenu fixé, alors que l'angle de pan P varie de manière symétrique, par exemple $2\,N + 1$ fois, depuis une position initiale en petits pas jusqu'à une position finale, c'est-à-dire :

$$P_i = P_{max}\,\frac{i}{N} \quad i \in \{-N, ..., N\}$$

– la fenêtre de lissage de l'image w qui définit le niveau de lissage de l'image et qui limite le niveau de bruit présents dans l'image à traiter.
– le seuil de détection de coin t qui permet d'éliminer les coins factices et les coins dont le contraste est insuffisant.

Les différentes étapes définies pour l'auto-calibration sont schématisée par le schéma de la figure 4.2.

4.1.2.2 Contraintes de mise en œuvre

Maintenant que nous avons présenté l'algorithmique de la tâche, nous allons étudier les contraintes de mises en œuvre.

Flot de données. Contrairement à l'élaboration de la carte de l'environnement, cette tâche d'auto-calibration est dépendante du capteur : les déplacements de la caméra ne sont pas prédéfinis comme au paragraphe 4.1.1, mais sont calculés en fonction de l'estimée f obtenue à la suite du processus de vision. Par conséquent, le résultat de la transformation du flot de

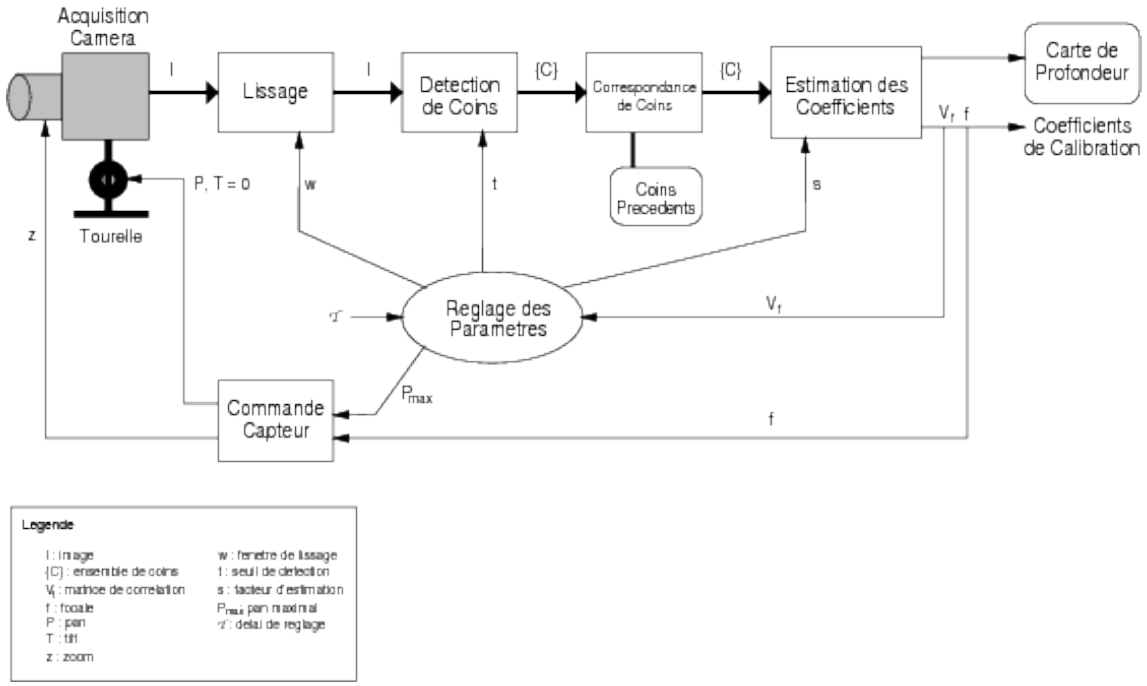


FIG. 4.2 – La séquence d'actions pour l'auto-calibration.

données n'est pas une image mais une *primitive image*, c'est-à-dire, une structure définie par une liste d'attributs, par exemple, les coefficients intrinsèques de calibration, ou un coin.

Le processus de calibration est une transformation cyclique dont la période dépend de l'évaluation des coefficients de calibration issus du calcul. Cette évaluation est faite à travers une fonction d'estimation qui décide de la validité du résultat en fonction par exemple du critère de qualité V_f.
Cette estimation peut être utilisée pour régler les paramètres de cette action, par exemple, le seuil de détection de coin peut être modifié pour augmenter le nombre de coins détectés au prochain cycle.

Par rapport à l'action décrite en 4.1.1, cette auto-calibration introduit de nouveaux types de modules algorithmiques :

- la transformation `im2tok`, c'est-à-dire, la transformation d'une image en un "token", par exemple, le module de détection de coins.
- la transformation `tok2tok`, c'est-à-dire, la transformation de "token" en une autre ensemble de token, par exemple, le module de mise en correspondance des coins.

Les opérations `im2tok` sont caractérisées en général par des traitements dont il est difficile de prévoir le nombre et la taille des données à traiter, car ce nombre dépend du traitement lui-même.
La même remarque s'applique aux modules `tok2tok` : ils mettent en œuvre des calculs symboliques sur des listes définies d'attributs mais plus sophistiqués que les modules `im2tok`. Par conséquent, la complexité n'est plus constante par rapport au nombre de données à traiter.

Paramètres. Cette action introduit un nouveau type de paramètre, que nous n'avons pas rencontré lors de l'action d'élaboration de la carte de l'environnement 4.1.1. Il s'agit d'un paramètre dont la valeur est produite par un mécanisme d'estimation/adaptation. Par exemple, en fonction de la valeur de V_f, qui est représentatif de la qualité de l'estimation de f, les paramètres t, w peuvent être modifiés lors de l'exécution nominale de l'action pour améliorer la qualité de l'estimée f au cycle suivant.

Les paramètres de cette action sont interdépendants. En effet, N dépend de P_{max} et de l'estimée, f, ce qui montre que le mécanisme d'adaptation/estimation de paramètres n'est pas simple à réguler.

De plus, ces paramètres influent non seulement sur la qualité de la transformation du flot de données, comme cela a déjà été dit précédemment au paragraphe 4.1.1, mais ils influent aussi sur la quantité de données transformées. Par exemple, lorsque P_{max} croît, le conditionnement numérique des équations est censé augmenter (étant donné que l'information visuelle concernant le mouvement est liée directement à la disparité dans une séquence d'image) mais de moins en moins de points vont être mis en correspondance le long de la séquence. Alors, le module de mise en correspondance de coins transforme moins de données en entrée mais la qualité de la valeur de f estimée risque de ne plus être suffisante.

Contraintes temporelles. L'auto-calibraton est réalisée cycliquement mais ne doit pas satisfaire de contraintes temporelles particulières. Le cycle dépend du résultat estimé, si il est jugé suffisamment juste l'action s'arrête sinon une nouvelle boucle de calcul est lancée.

Contrairement à l'élaboration de la carte de l'environnement, 4.1.1, le temps de calcul ne dépend plus de la quantité de données à traiter en entrée des modules algorithmiques. Il dépend du nombre de coins qui ont pu être détectés dans la scène observée. L'action d'auto-calibration montre l'influence directe de l'environnement sur la quantité d'informations à traiter et par conséquence sur le temps de calcul.

Capteurs. Cette action est très fortement liée au capteur visuel étant donné qu'elle a pour but de déterminer les coefficients intrinsèques de la caméra. Par conséquent, le contrôle du capteur est primordial.

Les mouvements de la caméra sont obtenus au travers d'une boucle de contrôle "fermée", contrairement à la boucle ouverte présentée dans la partie 4.1.1, c'est-à-dire, les nouveaux déplacements de la caméra dépendent des mouvements précédents et des calculs issus de ces mouvements. Donc, d'un côté, on doit gérer des mouvements dépendants du capteur, c'est-à-dire, les déplacements dépendent du matériel, d'un autre côté le contrôle logiciel des capteurs doit permettre d'assurer une interaction modulaire et transparente avec les autres modules algorithmiques qui vont interagir avec le capteur. Si le capteur est changé, on ne doit pas réécrire les modules algorithmiques pour que la même action puisse être réalisée. Par conséquent, le contrôle doit encapsuler les spécificités matérielles des capteurs.

Événements de contrôle. Si on considère les différentes phases définies précédemment dans la partie 4.1.1, les caractéristiques événementielles de l'auto-calibration sont les suivantes :

- Phase d'exécution nominale :

  - Événements internes : lors du cycle de calcul, en fonction de l'estimation qualitative du résultat, des paramètres peuvent être modifiés pour améliorer le prochain cycle de résultat. Cela sous-entend que la reparamétrisation de certains modules doit être lancée. Cette information de reparamétrisation peut être représentée par un signal envoyé aux modules concernés, mais on doit aussi envisager un mécanisme logique qui a pour charge de déterminer parmi les modules algorithmiques présents dans la chaîne de traitements ceux qui doivent être reparamétrés.
- Phase de fin :
  - Événements internes : en fonction de la valeur de V_f utilisée pour évaluer l'estimée f, l'action d'auto-calibration est soit relancée, si l'estimée n'est pas jugée suffisamment bonne ou arrêtée dans le cas contraire. Ceci fournit une condition de *bonne fin* pour l'action, condition issue de la chaîne de transformation elle-même.
    On peut définir aussi une condition de *mauvaise fin* générée dans le cas où aucun résultat suffisamment juste n'est obtenu.
  - Événements externes : l'action peut être préemptée à la suite d'une interaction extérieure à la chaîne de transformation algorithmique, par exemple, un événement reçu d'une interface utilisateur ou issu du capteur dans le cas où la caméra présente un problème.

4.1.3 Suivi de régions en mouvement

4.1.3.1 Principe algorithmique de la tâche

But : Détecter et suivre une cible en mouvement dans un environnement d'intérieur.

Comment :

1. **Acquisition et lissage** de l'image est réalisée en utilisant les opérateurs câblés sur la carte d'acquisition vidéo fournie pour la caméra. Le lissage aurait pu être fait en temps réel par contrôle du focus comme décrit dans [Viéville 95].
   Le zoom n'est pas utilisé car on veut travailler uniquement avec des rotations pures de la caméra.
2. **Stabilisation de la rotation**. La rotation de la caméra entre deux images successives est supposée être connue à partir des encodeurs de la tête du système de caméra. Une *transformation affine approximée* correspondant à la disparité de rotation est déterminée comme décrit par [Murray 93, Viéville 95].
3. **Différence d'image**. On calcule la valeur absolue de la *différence d'image stabilisée* entre une image au temps t et celle au temps $t-1$ (c'est-à-dire, l'image précédente sur laquelle la transformation affine a déjà été appliquée pour compenser la disparité de rotation) et on seuille cette valeur de telle manière à éliminer les disparités parasites dues aux erreurs de stabilisation ou au bruit dans l'image.
4. **Pondération autour de la fovéa**. Pour focaliser sur la fovéa de la rétine et aussi sur les objets en mouvements, on pondère la disparité résiduelle avec une courbe prédéfinie (dans notre cas, il s'agit d'une gaussienne) comme présenté dans la figure 4.4.

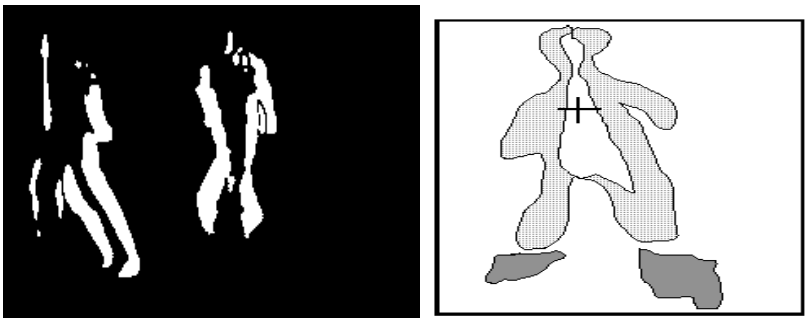


FIG. 4.3 – Seuillage basé sur la différence d'image stabilisée avec un seuil fixe. Les ombres peuvent venir perturber la localisation des pieds des cibles.

Cette pondération permet d'éluder les scintillements parasites dûes à l'éclairage d'intérieur situé en haut de l'image, mais aussi de *stabiliser* le suivi étant donné que les points suivis tendent à être collés à la fovéa de l'image. Cependant, cette stratégie limite bien évidement les capacités du système à passer à des cibles excentrés. De plus, si la pondération est trop étroite, l'estimation des cibles risque d'être biaisée et le suivi d'être décalé.

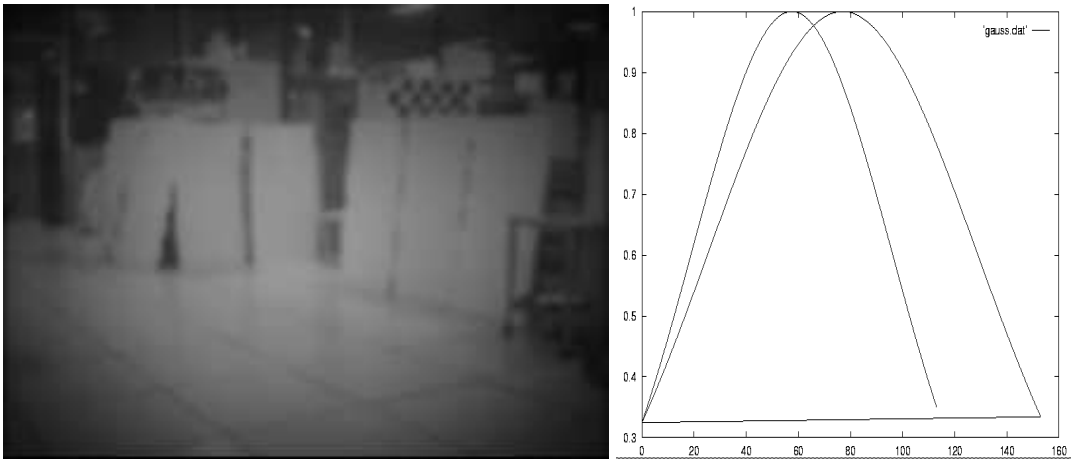


FIG. 4.4 – Pondération autour de la fovéa.

5. **Détections de régions**. On regroupe les pixels en *régions de mouvement rectangulaires* en utilisant un algorithme à balayage sur l'image.
   Le principe de cet algorithme est relativement simple et est illustré par les figures 4.5 et 4.6.
   – l'image est parcourue ligne par ligne et colonne par colonne afin de détecter des discontinuités entre une ligne/colonne sans pixel de mouvement et une avec pixel de mouvement. Ce balayage se fait en alternant le balayage vertical et horizontal de manière itérative sur chaque nouvelle zone détectée. L'algorithme prend fin lorsque le balayage sur une zone d'intérêt ne détecte pas de nouvelle région.

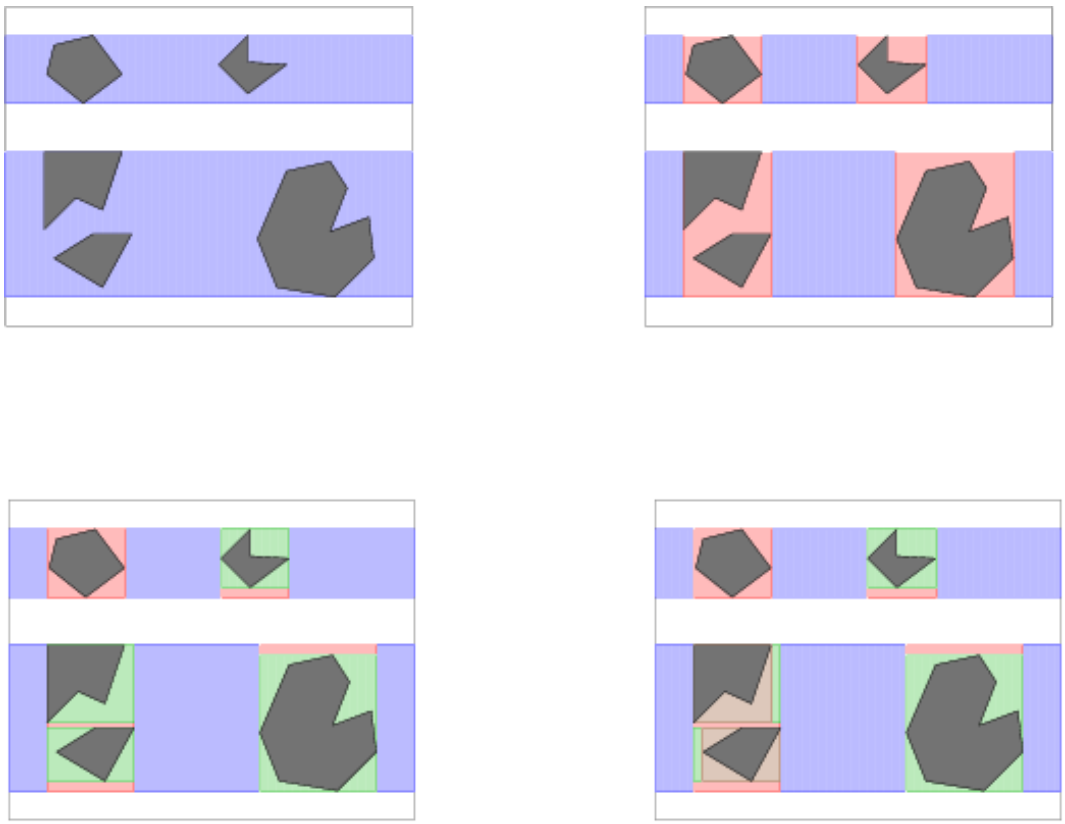


FIG. 4.5 – On commence à balayer l'image dans une direction, ici la direction horizontale et on réitère le processus pour chaque zone décelée. Ici, les zones issus du premier balayage sont en bleu, celles du deuxième balayage en rouge.

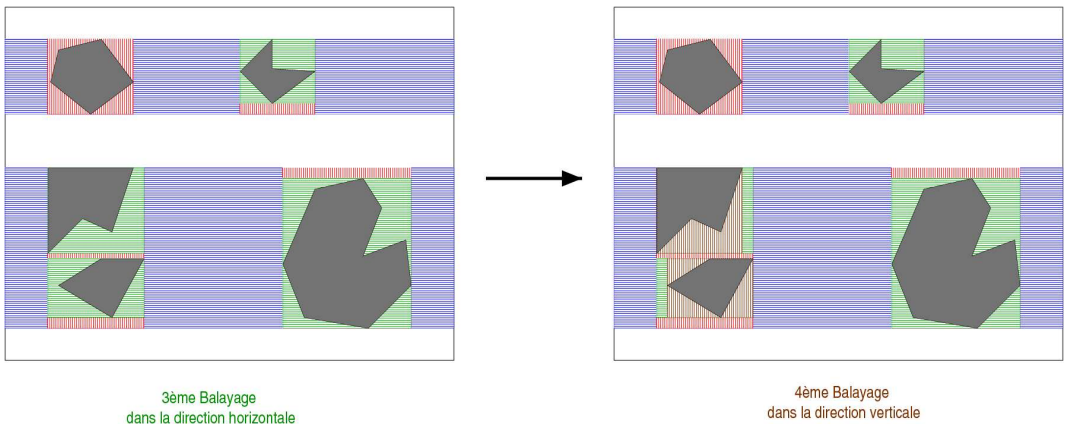


FIG. 4.6 – On continue le balayage tant que toutes les zones n'ont pas fini d'être explorées. Le troisième balayage est représenté en vert et le dernier balayage en brun.

Au final, l'algorithme permet de localiser les bords gauches/droits (par le balayage vertical) et supérieurs/inférieurs (par le balayage horizontal) de chacune des zones correspondant à des régions de mouvement.
Cependant, après la détection du mouvement, un objet en mouvement correspond en général à deux zones de mouvement[2] comme l'illustre la figure 4.7. Pour gérer cette particularité, le balayage est fait par un segment dont la largeur est paramétrée et représente l'écart entre ces deux zones de mouvement. Ainsi, l'algorithme permet de regrouper ces deux zones en une seule région, et la partie centrale est incluse dans la zone en mouvement comme cela est attendu.
Toutefois, la valeur du paramètre de largeur du segment de balayage influe sur la justesse de détection des régions de mouvement. Si la valeur est trop faible, on ne regroupe pas deux zones faisant partie de la même région de mouvement, et inversement, si elle est trop grande on regroupe des zones qui correspondent à des cibles différentes.

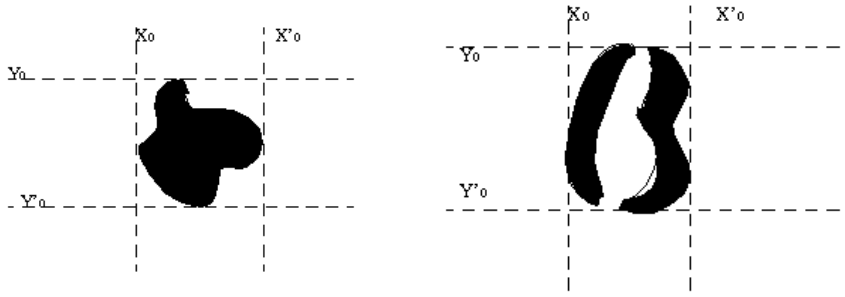


FIG. 4.7 – Contour rectangulaire d'une région de mouvement (à gauche), région de mouvement pour l'objet en mouvement (à droite).

– dans le cas où plusieurs objets bougent, une région peut en masquer une autre comme l'illustre la figure 4.8. Ce masquage est corrigé par des balayages successifs des lignes pour détecter les ($Y0$,$Y0^{'}$) et ($Y1$,$Y1^{'}$) séparément.

Cet algorithme de localisation des projections rétiniennes des objets en mouvement est rapide et simple. Cependant, il ne permet pas de détecter certaines configurations tels que l'exemple présenté à la figure 4.9.

La figure 4.10 présente un exemple des performances de cette approche, sachant qu'elle est suffisament efficace dans notre cas, car on s'intéresse à des scènes d'intérieur avec peu d'objets en mouvement.

6. **Mise en correspondance des régions**. D'une image à une autre, chaque région rectangulaire est mise en correspondance avec le meilleur candidat de l'image précédente. Ceci génère des séquences de "régions d'intérêt". Les petites régions liées le plus vraisemblablement à des artefacts sont éliminées en utilisant un *seuil de taille de région*.

[2]une zone correspond à la partie du fond masqué par l'objet et l'autre zone correspond à la partie du fond précédemment masqué qui se retrouve découverte.

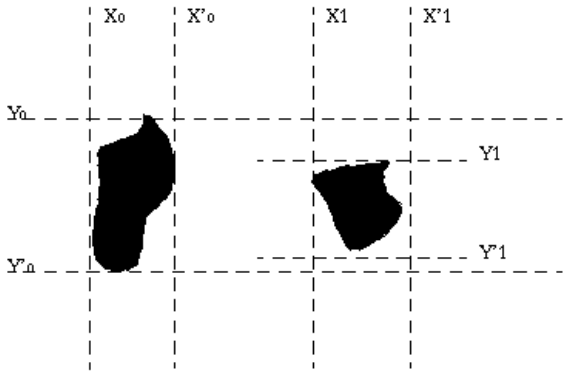


FIG. 4.8 – Intersection de régions quand deux objets sont détectés en même temps.

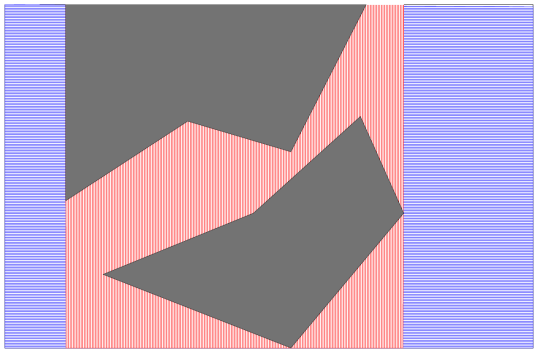


FIG. 4.9 – Configuration où l'algorithme de balayage ne permet pas de déceler deux zones distinctes.

Cette action est paramétrée par :

- la *fenêtre de lissage*, w, qui est utilisée lors du lissage de l'image acquise par le capteur. w est donné en pixel (typiquement 2 à 16 pixels). Cette valeur dépend de la nature de la scène observée.
- le *seuil de variation d'intensité*, t, qui est utilisé lors de la différence d'image. Il varie entre 10 et 40, la valeur typique étant de 16, lorsque l'on travaille sur des images en 256 niveaux de gris. La valeur $t = 16$ a été validée empiriquement.
- la *taille moyenne d'une zone*, a, qui est utilisée lors de la pondération. Ce paramètre varie en fonction de la dynamique de la cible suivie. Plus précisément, il doit être réglé de façon à *minimiser l'erreur de suivi visuel au cours du temps*.
- l'*intervalle entre deux régions*, g, qui est introduit lors du regroupement des régions pour gérer le cas où, l'image ayant été lissée et seuillée, certaines parties peuvent être gommées si un intervalle d'un pixel existe dans le profil original d'intensité, comme illustré à la figure 4.11.
  Donc, si deux régions adjacentes sont telles que leur distance verticale et horizontale est plus petite que l'intervalle g, elles sont regroupées en une seule et même région.
  Par conséquent, il devient simple de déterminer la valeur de l'intervalle à partir des paramètres w, t et l'amplitude moyenne de la disparité dans une région A. Finale-

FIG. 4.10 – Exemples d'objets en mouvement : ici, l'action détecte grossièrement la zone comprenant les objets. En bas de chaque fenêtre, l'intersection de la fenêtre avec le sol entraîne une sous estimation de la valeur de la profondeur, avec une erreur inférieure à 50 cm.

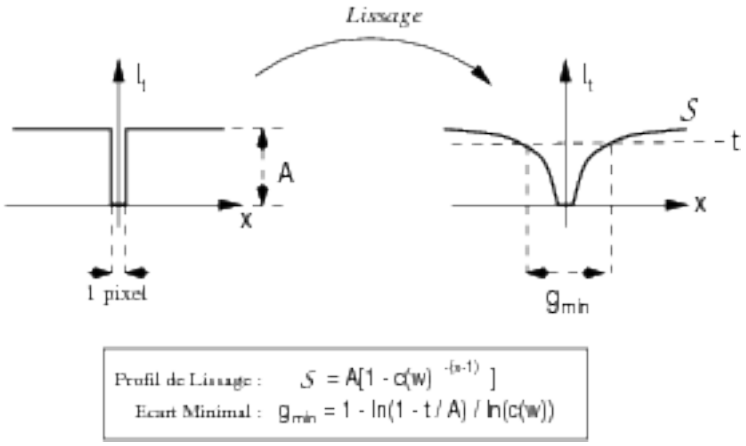


FIG. 4.11 – Évaluation de l'intervalle entre deux régions consécutives suite au lissage et seuillage de l'image. Ici les formules correspondent à un filtre récursif exponentiel (récursif au premier ordre) qui a été utilisé. Le paramètre c du filtre est lié directement à la fenêtre de lissage w.

ment, on a choisi $g = 2\,g_{min}$, c'est-à-dire, deux fois cette valeur pour être dans le bon côté.

– le *seuil de taille de région*, s, qui est introduit lors de la mise en correspondance des régions. Il permet d'éliminer les *petites régions* vraisemblablement liées à des artefacts.

Les différentes étapes effectuées pour le suivi de régions sont schématisées par la figure 4.12.

4.1.3.2 Contraintes de mise en œuvre

Flot de données. Cette action produit différents types de résultats : une *primitive*, c'est-à-dire, la région suivie, une commande pour le capteur visuel et une *image*, étant donné qu'un utilisateur peut vouloir visualiser la scène observée et la région suivie.

Le flot de données est périodiquement transformé et la transformation doit satisfaire des contraintes temps réel. Quelque soit la quantité de données à traiter, la suite de calculs doit se faire en un temps limité, et doit fournir les trois types de résultat à chaque fin de cycle.

Paramètres. Par rapport aux actions analysées aux paragraphes 4.1.1 et 4.1.2, le suivi de régions en mouvement utilise des paramètres qui peuvent être assignés interactivement tout le long de la transformation du flot de données, par exemple, le seuil s de taille de région en mouvement.

En plus de l'interdépendance des paramètres déjà rencontrée au paragraphe 4.1.2, les paramètres mis en jeu ici sont très fortement tributaires de l'application. Leurs valeurs sont liées à la nature de la scène observée, et des objets de l'environnement.
Par exemple, le seuil sur la taille des régions à suivre s varie suivant que l'on suit un géant ou un nain !
On doit trouver des régions en mouvement stables correspondant le plus vraisemblablement à de vrais objets en mouvement. La performance de cette détection peut être quantifiée par *le nombre de régions en mouvement* qui doit être minimal si une région particulière en mouvement englobe un objet et non pas des ombres parasites ou des ensembles d'objets. Les paramètres w et t doivent être ajustés de façon à optimiser ce critère, lié à l'observation empirique des séquences de suivi.

Comme cela a déjà été mentionné dans le paragraphe 4.1.2, les paramètres influent sur la quantité de données à traiter en entrée des modules algorithmiques, et par conséquent, le temps de calcul des modules peut varier. Donc, le respect des contraintes temporelles est relié aux valeurs des paramètres. Donc, l'adaptation des paramètres peut être gérée pour que la transformation du flot de données satisfasse les contraintes temporelles.

Contraintes temporelles. Le flot de données est périodiquement transformé et des contraintes temps réel doivent être satisfaites. En effet, si l'action s'exécute avec une période plus grande que la dynamique des objets, on ne pourra rien détecter. Aussi, il faut pouvoir limiter le temps de calcul et il faut fournir un mécanisme pour jouer sur la quantité de données à transformer. Si la chaîne de transformations ne peut pas être traitée dans la durée du cycle, il est important de pouvoir détecter ce dépassement. Ensuite, il faut pouvoir décider de la politique la plus appropriée pour résoudre ce non respect des contraintes temporelles. Cette politique définit comment contraindre au mieux la quantité de données à traiter au prochain cycle pour éviter un nouveau dépassement. Par conséquent, un mécanisme de détection, décision et adaptation doit être fourni. Regardons les besoins pour ces trois entités.

– Comment détecter un dépassement de période ?

1. Dans le pire des cas, lorsqu'il survient ! Mais alors la chaîne de traitements ne produit aucun résultat ou alors un résultat intermédiaire.
2. Si chaque module connaît le temps restant avant la fin du cycle, et son propre temps de calcul, il serait possible de déduire si il peut ou non achever son traitement dans le laps de temps restant. Pour arriver à cela, il faudrait fournir à chaque module algorithmique des informations supplémentaires concernant le temps de calcul, la période du cycle de traitement, etc.

3. On peut envisager de disposer d'une entité décisionnelle qui connaît, pour chaque module algorithmique de la chaîne, le temps minimal et maximal de calcul ainsi que le délai avant la fin du cycle. Alors, cette entité est capable de déterminer le temps restant après l'exécution de chaque module, et de décider si les modules suivants dans la chaîne doivent être lancés dans un mode dégradé ou non en fonction du temps estimé.
   Le problème, ici, est d'obtenir pour chaque module son temps minimal et maximal de calcul. La plupart des algorithmes de type (image $\rightarrow$ image) utilisés en vision ont une complexité qui dépend linéairement de la quantité de données à traiter : par exemple, le lissage de Sobel. Mais ce n'est plus vrai pour les algorithmes basés sur des considérations stochastiques, ou plus généralement ceux qui se fondent sur une boucle de convergence. Dans ce cas, on peut fixer un nombre prédéfini d'itérations mais la justesse du résultat n'est plus assurée.

– Décision : en fonction des précédents moyens de détections des dépassements de période, on peut envisager les stratégies suivantes pour y remédier.
   1. Une première stratégie consisterait à garder une copie des résultats du cycle précédent et de le restaurer au cycle courant en cas de dépassement des délais.
   2. Dans le cas où un module n'a pas pu exécuter son traitement entièrement, il pourrait passer en mode dégradé. Dans ce mode, le module limite son temps de calcul et donc limite le nombre de données à traiter.
      Ce mode dégradé est dépendant de l'action dans laquelle intervient le module algorithmique et il doit être spécifié par le concepteur de l'action. Ce mode dépend des calculs et de la nature du résultat des calculs des modules (commande robotique, information symbolique, etc.). Par exemple, en mode dégradé, les modules peuvent fournir une copie de la sortie du cycle précédent, une valeur nulle ou encore l'entité de décision peut demander d'arrêter les modules de la chaîne de traitement restant à exécuter et à fournir, comme résultat du cycle de traitement, le résultat du cycle précédent.
      On peut s'attendre à ce que, si un module passe en mode dégradé, les prochains modules dans la chaîne de traitement puissent s'exécuter avant la fin du cycle.
      Mais la connaissance du temps de calcul est locale à chaque module : un module ne sait rien des autres modules participant à la chaîne de traitement et on ne peut pas établir de stratégie globale à tous les modules.
   3. Une troisième stratégie consisterait à choisir un autre module ayant la même fonctionnalité mais avec un temps de calcul plus petit. Ceci nécessite de disposer d'une bibliothèque de fonctionnalités et de pouvoir interchanger facilement, d'un cycle à un autre, des modules de même fonctionnalité.

– Adaptation : pour assurer que le prochain cycle de traitement satisfera les contraintes temporelles sans changement de mode, il serait intéressant d'utiliser une méthode basée sur l'estimation/adaptation de paramètres qui permettrait aux modules de la chaîne de traitement de contraindre la quantité de données à traiter au prochain cycle.
   Par exemple, en augmentant le seuil t utilisé lors de la différence d'image, on détecte moins de régions en mouvement. Donc, les modules suivants dans la chaîne de traitement vont manipuler moins de données et par conséquent, les calculs seront moins

gourmands en ressource temps et inversement.
Une autre solution consiste à fixer un nombre maximal de zone en mouvement.

Capteurs. L'acquisition d'image du capteur visuel se fait suivant la cadence imposée par les contraintes temporelles du suivi. On peut envisager une cadence de 25 images par seconde, ceci sous-entend qu'une nouvelle image serait délivrée toutes les 0.04 secondes.

Le cycle de traitement comprend à la fois la transformation de flot de données mais aussi le temps de l'acquisition et de déplacement de la caméra. Ceci implique donc que le capteur doive aussi satisfaire des contraintes temps réel bien précises.

Événements de contrôle. On peut définir les événements suivants pour l'action de suivi.

- Phase d'initialisation :
  - Événements externes : étant donné qu'il n'est pas utile de faire de suivi lorsque la scène à observer est vide, cette action a besoin de la réception d'une condition particulière pour être activée. Par exemple, "du mouvement à détecter dans la scène". Cette condition peut aussi être émise par un utilisateur ou par une autre action dont le but est de détecter du mouvement dans la scène. Ce dernier cas illustre le besoin de séquencer deux actions faisant partie d'une application complexe.
- Phase d'exécution nominale :
  - Événements internes : dans le paragraphe concernant les paramètres, nous avons expliqué que l'on peut jouer sur les paramètres pour que la transformation du flot de données respecte ses contraintes temporelles. Donc, nous avons besoin de signaux de reparamétrisation (ou événements) liés à une entité qui vérifie le temps et qui vont être envoyés pour notifier le changement de paramètres aux modules algorithmiques concernés.
    Dans ce cas, la notification n'est pas simple à mettre en œuvre, car elle doit viser les paramètres les plus appropriés. Ceci implique, donc, de pouvoir caractériser précisément chaque paramètre adaptable.
    Une autre façon d'assurer le respect des contraintes temporelles est de changer un module algorithmique contre un autre de même fonctionnalité et de disposer d'un critère de sélection en fonction de chaque fonctionnalité pour choisir le module le plus approprié par rapport aux besoins temporels à satisfaire.
  - Événements externes : comme cela a déjà été décrit pour l'action d'auto-calibration (paragraphe 4.1.2), la reparamétrisation des modules algorithmiques peut être faite à travers d'une interface utilisateur ou un processus extérieur.
- Phase de fin :
  - Événements internes : comme pour l'action de l'auto-calibration, on peut définir une condition de fin pour le suivi. En effet, lorsque plus aucun objet n'est en mouvement, il n'est plus utile de continuer à faire du suivi. Pour ce faire, on peut définir un critère qui décidera du moment où le suivi doit prendre fin. Par exemple, si au bout de x cycles, on ne détecte plus de mouvement, le suivi s'achève.
  - Événements externes : l'action peut être interrompue comme dans le cas décrit dans 4.1.2 par une interaction externe, telle que l'intervention d'un utilisateur, un signal d'urgence (par exemple, "caméra KO", "grand-mère est tombée", "arrêt de l'utilisa-

teur").
Il est important de noter qu'un signal d'urgence ne concerne pas seulement l'action en cours. En effet, elle concerne aussi les autres actions qui prennent part à une application complexe. Alors, toutes les actions composant l'application et actives doivent être stoppées (ceci sera décrit plus en détail au chapitre 5).

4.2 Synthèse : caractéristiques d'une application de vision

4.2.1 Définition de la *tâche élémentaire* de vision

Les trois tâches analysées aux paragraphes 4.1.1, 4.1.2 et 4.1.3 illustrent les chaînes de traitements types qui entrent dans la composition des applications de vision temps réel qui nous intéressent. Elles introduisent de manière incrémentale les spécificités des traitements mis en jeu dans ces applications.

Pour mettre en œuvre une tâche, l'analyse a montré qu'il faut prendre en compte les aspects gestion du capteur, la gestion des paramètres, le contrôle et les contraintes temporelles, en plus de la partie algorithmique, qui transforme successivement le flot de données issu à l'origine par un capteur.

Etudions maintenant les caractéristiques que l'on peut déduire de ces trois analyses, en fonction des différentes contraintes de mise en œuvre que nous avons extraites.

Transformations algorithmiques. Les tâches mises en œuvre correspondent à une chaîne de transformations algorithmiques sur un flot de donnée issu d'un capteur. Chacune de ces transformations implantent des algorithmes de traitements d'image ou de vision. Cet ensemble d'algorithmes peut être mis sous la forme de schémas-blocs fonctionnels (c'est-à-dire, des unités d'organisation de programme, instanciables, retournant un ou plusieurs éléments de données après son exécution). Ce schéma exprime comment **le flot de données** est manipulé par chaque bloc, depuis l'image source jusqu'au résultat final. De plus, la chaîne des transformations que nous considérons doit être réalisée périodiquement et doit respecter des contraintes temps réel particulières. En effet, ces transformations doivent être faites suivant la cadence imposée par la dynamique de la scène observée.
En fonction des données en entrée et des données produites, nous distingons trois types d'algorithmes :

i. image vers image, notée [`im2im`] : par exemple, le lissage d'une image.
ii. image vers token, notée [`im2tok`] : par exemple, la détection de coin.
iii. token vers token, notée [`tok2tok`] : par exemple, le "matcher de coins".

Suivant le type d'algorithme, la complexité en terme de quantité de données et la complexité en temps sont différentes. Par exemple, la plupart des `im2im` sont linéraires par rapport aux données d'entrée.

Les paramètres. Les transformations sont conditionnées par un flot de données discontinu particulier : les paramètres. Ces paramètres servent de régulateur du flot de données transformées dans l'image. En effet, ils permettent de conditionner la qualité des résultats

ainsi que la quantité d'informations issue des traitements algorithmiques. Par exemple, en fonction de la valeur d'un seuil, on peut limiter le nombre de segments à détecter. Dans le domaine de la vision, les paramètres peuvent être interdépendants et leur valeur dépend fortement du contexte applicatif de la chaîne de traitements : par exemple, un seuil de lissage peut dépendre de la luminosité d'une pièce. Ces valeurs peuvent être :

- fixées une fois pour toute ou fixées à l'initialisation des traitements ;
- positionnées de manière interactive à travers un processus de type interface homme-machine ou un autre processus extérieur à la chaîne de traitement ;
- adaptées en fonction de l'évaluation de la qualité des résultats issus de la chaîne de transformation (amélioration de la qualité de la transformation) et pour satisfaire les contraintes temporelles (modification du débit du flot de données).

Contrairement au flot de données, qui est transformé périodiquement, la fréquence de leur mise à jour peut dépendre de l'évaluation faite sur le flot de données.

Les capteurs. Ils constituent l'interface entre l'environnement et la chaîne de transformation du flot de données.
Ils assurent la production de la source du flot de données qui va être transformé. De plus, ils peuvent recevoir des commandes issues de traitements de la chaîne de transformations. En conséquence, ces capteurs doivent être contrôlés pour assurer que la boucle entre l'acquisition des données et les commandes soit réalisée en respectant les contraintes temporelles du système. Pour ne pas rendre la chaîne de traitement dépendante d'un capteur particulier, son contrôle doit être assuré indépendamment de ses spécificités matérielles.

Les contraintes temporelles. Les applications qui nous intéressent sont de type temps réel, c'est-à-dire qu'elles doivent s'exécuter en respectant des contraintes de temps. Dans notre contexte de travail, ces contraintes de temps sont imposées par la dynamique de la scène à observer : en général on souhaite satisfaire la cadence video (25 images par seconde).
Par conséquent, la chaîne de transformations doit s'exécuter entièrement avant l'échéance de la période. Il faut donc pouvoir contrôler les temps de calcul des différents traitements algorithmiques. Les traitements à envisager doivent s'exécuter en un temps qui peut être majoré. On a vu avec les paramètres qu'une façon de faire est d'ajuster le débit du flot de données à transformer.

Le contrôle. Le contrôle est défini comme une séquence d'états liée aux phases logiques de la transformation du flot de données (initialisation, calcul nominal, fin). On passe d'un état à un autre sur l'occurence d'événements logiques particuliers à la transformation. Le *comportement logique* associé à la transformation globale du flot de données est entièrement spécifié par l'ensemble des séquences permises d'événements logiques d'entrée et sortie qui constituent l'interface de la transformation avec son environnement. Ce comportement logique intègre à la fois le contrôle externe et interne nécessaire à la tâche de vision :

- en interne : les phases logiques (initialisation, exécution nominale, terminaison) de chaque sous-module, qui implante le flot de données, doivent être encapsulées au sein d'un schéma uniforme et générique. C'est à ce niveau que, lors de l'exécution nomi-

nale, les changements de mode d'exécution des sous-modules en fonction d'événements internes sont contrôlés (par exemple, un module doit être repamétré).

- en externe : lorsque l'on construit une application complexe, plusieurs tâches de vision doivent être coordonnées les unes par rapport aux autres, c'est-à-dire il faut des mécanismes de démarrage et d'arrêt de tâches ainsi que des mécanismes de notification sur l'exécution.

Nous synthétisons dans le tableau 4.1 les différents points que nous venons d'exposer.

TAB. 4.1: Les caractéristiques de la tâche de vision

`Transformations`	➟ `source` : capteur de vision
	➟ `résultat` : image, token, commande d'un capteur, information qualitative
	➟ séquence de transformation temps réel périodique ou non
	➟ transformation paramétrée
	➟ `type de transformation :` `im2im`
	`im2tok`
	`tok2tok`

`Capteurs`	➟ interface entre l'environnement et la transformation du flot de données ➟ acquisition + boucle de contrôle temps réel ➟ monitoring obligatoire ➟ contrôler indépendament des spécificités matérielles du capteur
`Paramètres`	➟ `source` : interne ou externe à la transformation du flot de données ➟ régulateurs du flot de données ➟ débit discontinu par rapport au débit de flot de données ➟ conditionnent la qualité & la quantité issues de la transformation du flot de donnée ➟ interdépendants ➟ dépendants de l'application ➟ peuvent être adaptés pour satisfaire les contraintes temporelles ➟ `valeur du paramètre :` fixe déterminée à l'initialisation de la tâche de vision adaptée en fonction de l'estimation du résultat de la transformation positionnée de manière interactive ou par un processus extérieur
`Contraintes Temporelles`	➟ dépendent de la dynamique de la scène ➟ temps de calcul imposé par la dynamique de l'application ➟ temps de calcul contrôlable par ajustement du débit du flot de données à traiter

TAB. 4.1: Les caractéristiques de la tâche de vision

`Contrôle`	➟ phases de [`initialisation`, `exécution`, `fin`] conditionnées par des événements logiques ➟ `type d'événements` : interne & externe à la tâche de vision ➟ `conditions d'initialisation` événements internes : initialisation des structures de données événements externes : conditions nécessaires au lancement de la tâche ➟ `pendant la phase d'exécution` événements internes : événements provenant des capteurs reparamétrisation liée à l'évaluation de la qualité d'un résultat reparamétrisation pour assurer le respect des contraintes de temps événements externes : reparamétrisation faite par un processus extérieur (e.g. IHM) ➟ `conditions de fin` événements internes : BONNE/MAUVAISE FIN événements externes : PRÉEMPTION de la tâche de vision événement externe d'alerte ➟ comportement logique = séquence(état, événements) ➟ comportement logique modélise le contrôle de la tâche de vision

Cette synthèse nous permet de déduire une caractérisation de l'entité de base qui va nous servir pour la spécification des applications de vision. Nous appelons une telle entité la *tâche de vision élémentaire*. Cette tâche élémentaire va se baser sur le formalisme des modèles hybrides, dans le sens où elle va contenir à la fois des aspects algorithmiques et contrôle.

Nous définissons une **tâche élémentaire de vision** comme :
- ➟ *une chaîne périodique de **transformations paramétrées** appliquées à l'information visuelle fournie par un **capteur**.*
- ➟ *un contrôle qui supervise au **niveau logique** l'exécution de la chaîne de transformations pour gérer le respect des **contraintes temporelles** et assurer la conformité du résultat de la transformation.*

4.2.2 Spécification hiérarchique d'une application

Jusqu'ici nous avons vu comment spécifier une action élémentaire en utilisant le formalisme présenté au paragraphe 4.2.1. Spécifier des actions c'est bien, mais spécifier une application complexe, telle que l'application de la "mamy-sitter" 2.1, c'est mieux ! Pour cela, il faut qu'au niveau application on puisse définir le contrôle sur les tâches de vision. Celui-ci ne se spécifie pas simplement comme une concaténation d'actions. Ce contrôle s'exprime plutôt comme la coordination d'un ensemble d'actions : c'est-à-dire un ensemble d'actions mises en *séquence*, et/ou en *parallèle*, qui ont besoin d'être *synchronisées* les unes par rapport aux autres et qui peuvent être *interrompues* suite à la terminaison d'une autre action ou suite à une défaillance pendant l'exécution nominale, ou encore par une intervention extérieure.

Par conséquent, cette coordination nécessite un ordonnancement logique et temporel des tâches de vision :
par exemple, tâche X `puis` tâche Y `tant que STOP non reçu alors boucle` tâche W `en parallèle avec` tâche Z `jusqu'à URGENCES si URGENCES alors` ALARME.
Cet ordonnancement définit le contrôle global sur l'application. Ce contrôle logique doit être basé sur les caractéristiques logiques des tâches de vision.

Par conséquent, pour chaque tâche de vision, un sous-ensemble de caractéristiques est extrait du comportement logique pour offrir des points d'entrée de coordination du contrôle de la tâche de vision. Ce sous-ensemble est appelé *vue abstraite* de la tâche de vision. Cette vue abstraite constitue l'interface de la tâche de vision.

Nous proposons d'exprimer le contrôle global de l'application et le contrôle inhérent à chaque tâche de vision selon le même formalisme. C'est une particularité importante car le contrôle, au niveau de l'application et au niveau de la tâche de vision va être manipulé de manière uniforme, ce qui va faciliter l'analyse logique de l'application globale.

4.2.3 Proposition de méthodologie

Nous proposons une méthodologie de programmation des applications de vision qui se fonde sur deux niveaux d'abstraction :

1. le niveau *tâche elementaire (tâche de vision)*. Ce niveau fournit un ensemble de tâches de vision qui sont toutes encapsulées par un schéma logique commun, et qui fournit des vues abstraites. Cet ensemble peut être organisé en bibliothèques ou "package" logiciel pour faciliter l'utilisation de ces tâches.
2. le niveau *application*. Ce niveau est utilisé pour spécifier un arrangement logique de tâches de vision qui va permettre de concevoir une application de vision donnée.

Par conséquent, lorsque l'on doit gérer une application complexe telle que l'application de la "mamy-sitter", la première étape consiste à exprimer l'application sous la forme d'entités manipulables par l'architecture définie plus haut. Plus précisement, lors de la première étape, (i) on va identifier les chaînes de transformations périodiques représentatives de chaque tâche de vision ; (ii) pour chaque chaîne, on va décider de l'algorithme à employer et (iii) on va définir le contrôle associé. Lors de la seconde étape, on va spécifier l'application en arrangeant les tâches de vision choisies à la première étape. Cet arrangement va utiliser les vues abstraites extraites de chacune des tâches de vision qui vont composer l'application.

La figure 4.14 explicite la démarche pour spécifier l'application de la "mamy-sitter" avec ce choix d'architecture à deux niveaux.

Au chapitre 3, nous avons présenté parmi les différents environnements et méthodologies, le système ORCCAD/MAESTRO qui repose également sur une architecture à deux niveaux de spécification, assez proches de nos prérogatives. Aussi nous est-il apparu intéressant d'analyser comment utiliser le système ORCCAD pour développer les applications de vision qui intéressent notre problématique.

Ainsi, au chapitre 5, nous allons présenter en détail la structuration du système ORCCAD/MAESTRO ainsi que la méthodologie de programmation associée.
Ensuite nous illustrons, au chapitre 6, comment nous avons spécifié, validé et implanté une première version de l'application de la "mamy-sitter" en utilisant la méthodologie et l'environnement ORCCAD. Cette mise en œuvre nous a permis de mettre en lumière certaines limites du système ORCCAD vis-à-vis des spécificités des applications qui nous intéressent, car cette architecture a été spécialement conçue pour répondre aux besoins spécifiques des applications robotiques.
Ces limites nous conduisent, au chapitre 7, à analyser l'adéquation d'ORCCAD avec la vision temps réél et à proposer des solutions pour pallier aux manques de cet environnement par rapport aux besoins des applications qui sont abordées dans ce travail.

FIG. 4.12 – La séquence d'actions pour le suivi de régions en mouvement.

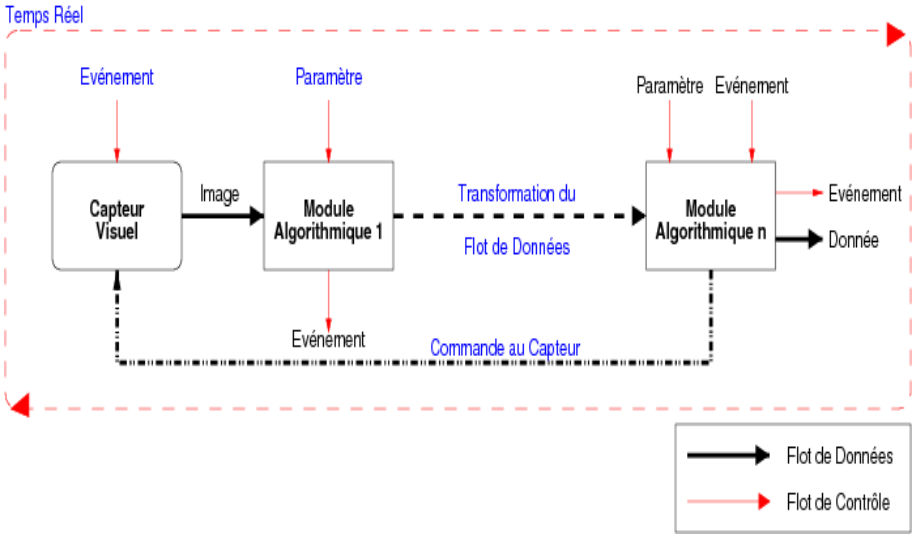


FIG. 4.13 – Représentation d'une TÂCHE VISION

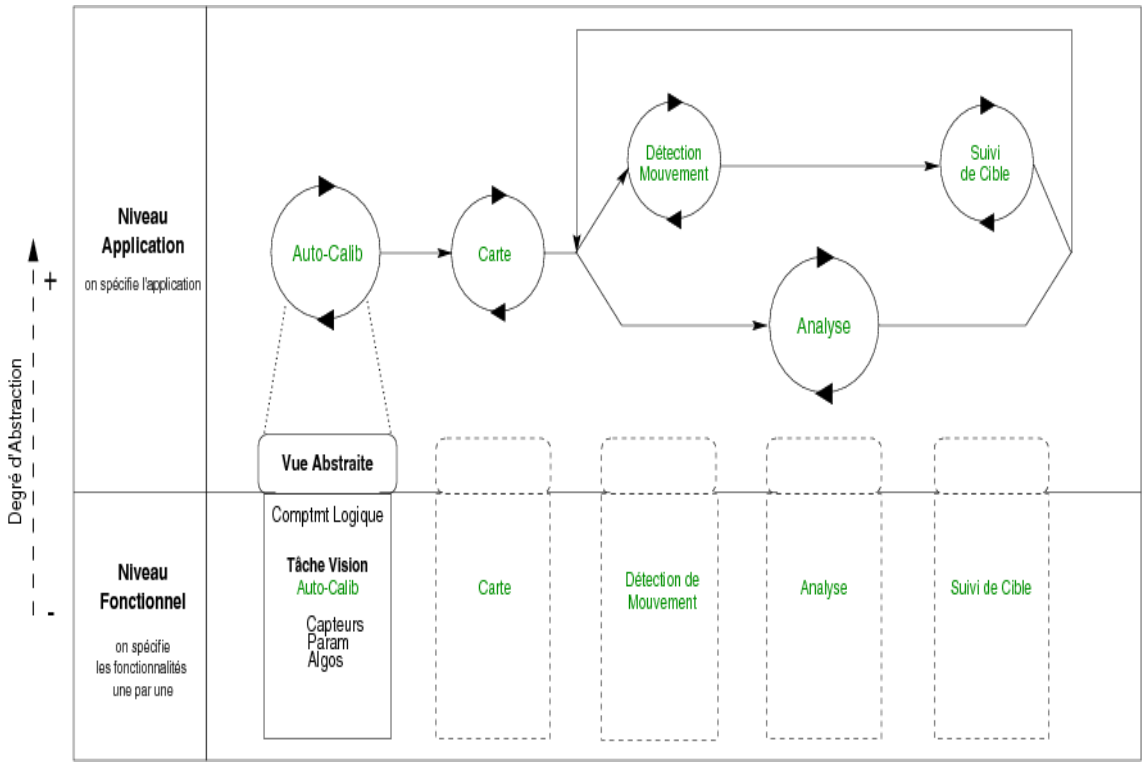


FIG. 4.14 – Hiérarchie de spécification proposée

Chapitre 5

L'architecture ORCCAD/MAESTRO

Comme nous l'avons vu au paragraphe 4.2.3, nous proposons une hiérarchie de spécification des applications de vision temps réel à deux niveaux :

- le niveau élémentaire qui manipule le "formalisme" de TÂCHE VISION. Cette tâche requiert la gestion d'une boucle temps réel à l'exécution et nécessite la gestion d'événements qui peuvent modifier son fonctionnement nominal.
- le niveau application qui va manipuler de manière **logique** un arrangement de TÂCHES VISION pour spécifier l'application globale.

Au chapitre précédent, 4.2.3, nous avons proposé de gérer ces deux niveaux par l'architecture hiérarchique ORCCAD/MAESTRO pour développer les applications de vision qui intéressent notre problématique.

ORCCAD (**O**pen **R**obot **C**ontroller **C**omputer **A**ided **D**esign) [Team 98] [ORCCAD WEB Page], désigne à la fois une méthodologie de programmation d'applications robotiques, et un environnement de programmation qui permet de mettre en œuvre cette méthodologie. Il permet de spécifier, valider et implanter une mission robotique complexe. Ce système s'inscrit dans le cadre des architectures de type "hybride" et est issu des travaux du laboratoire robotique de l'INRIA GRENOBLE et des équipes de recherche ICARE [ICARE WEB Page], BIP [BIP WEB Page], et CHIR [CHIR WEB Page], de l'INRIA, spécialisées dans le domaine robotique.

Les applications robotiques visées par cet environnement correspondent à des systèmes critiques tant au niveau temps réel que sur le plan de la *sûreté* de fonctionnement. En effet, ces systèmes interagissent fortement avec leur environnement à travers des actionneurs et des capteurs. De plus, la commande du robot doit être assurée périodiquement suivant des contraintes temporelles fortes pour assurer sa stabilité, et d'autre part, il faut prévenir toute panne ou dysfonctionnement dans la mission à remplir par l'application.

L'objectif clairement établi d'ORCCAD est de permettre *l'implémentation la plus fidèle possible des lois de commandes issues de la théorie de l'automatique*. Pour ce faire, ORCCAD propose à l'utilisateur des outils dédiés qui facilitent la programmation d'une application robotique. En particulier, il permet de valider la logique du système robotique en se basant sur des méthodes formelles et la mise en œuvre temps réel de l'application est assurée par génération automatique de code correspondant à la plateforme d'exécution. Ainsi, l'utilisateur se voit déchargé de la gestion explicite du temps réel et il peut alors se consacrer à l'élaboration de la commande robotique proprement dite et à l'élaboration du scénario de la

mission.

ORCCAD a été utilisé pour contrôler des systèmes tels qu'un manipulateur mobile (ROMEO et JULIET [Khatib 96], [Turro 99]), un véhicule électrique [Gusev 98], un robot sous-marin (le VORTEX [Kapellos 97]) ou encore un robot mobile (ANIS [Pissard-Gibollet 95]).

Les principes de cette architecture ont été largement détaillés dans [Team 98], mais nous allons présenter les caractéristiques principales de cet environnement et montrer comment on peut l'utiliser pour les applications de vision qui nous intéressent.

5.1 ORCCAD : une méthodologie de programmation

Au niveau méthodologie, ORCCAD aborde la programmation d'applications robotiques suivant l'approche "*contrôle commande*".

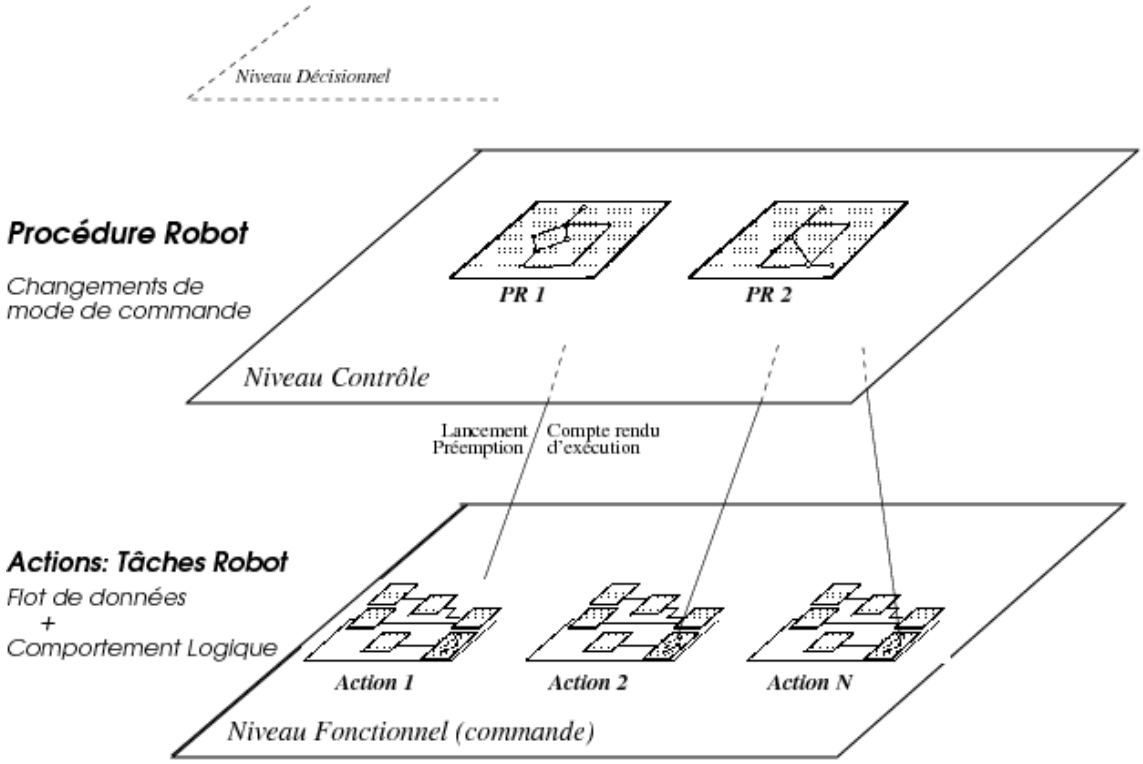


FIG. 5.1 – L'architecture ORCCAD

La méthodologie ORCCAD s'articule suivant deux niveaux d'abstraction : le *niveau fonctionnel* (ou commande) et le *niveau contrôle*.
Le *niveau fonctionnel* va fournir des actions élémentaires qui fournissent la commande robotique, le *niveau contrôle* va les combiner de manière logique pour constituer des applications.

Cette approche permet donc d'aborder la programmation d'applications complexes de façon incrémentale en fournissant une méthodologie de décomposition de l'application.
C'est le niveau fonctionnel qui va gérer les calculs sur des données numériques (ou plutôt sur

un flot de données) ; le niveau contrôle va manipuler des événements discrets (qui constituent le flot de contrôle du niveau données).
Ces deux niveaux communiquent entre eux par des *événements discrets*. Ces événements permettent de lancer les actions, de les interrompre (communication dans le sens : contrôle → commande), informent de l'état de l'exécution des actions (communication dans le sens : commande → contrôle), par exemple, bonne fin, anomalie dans l'exécution, etc.
Cette méthodologie distingue donc clairement la gestion d'un flot de données continu, les données numériques utilisées pour calculer une commande robotique (au niveau commande) et la gestion d'un flot de contrôle de nature discontinue, défini par l'intermédiaire d'événements discrets.

5.1.1 Le niveau fonctionnel

Ce niveau manipule une entité, appelée **Tâche Robot** (ou TR) qui est définie comme suit [Kapellos 94], [Turro 99] :
Une Tâche Robot est la spécification complète et paramétrable de :

1. la *ressource physique*, qui représente le système mécanique commandée par l'action.
2. d'une action élémentaire d'asservissement, c'est-à-dire d'une *loi de commande* de structure invariante sur la durée de la TÂCHE ROBOT.
3. du *comportement logique* associé à un ensemble de signaux concernant l'action élémentaire d'asservissement et susceptibles d'être émis préalablement à et pendant son exécution.

Par conséquent, ce niveau considère à la fois des aspects continus par l'intermédiaire de la loi de commande et des aspects discrets à travers le comportement logique.
La figure 5.2 représente en image le concept de TÂCHE ROBOT

La loi de commande. Elle traduit comment réaliser l'action robotique, i.e. elle exprime comment produire (et ceci à travers une boucle fermée) une commande à envoyer périodiquement aux actionneurs du robot à partir, par exemple, de mesures relatives au robot provenant de la ressource physique.
A titre d'exemple, voici, dans le cas d'un bras manipulateur robotique, une loi de commande **simple** permettant de bouger le bras d'une position courante à une position désirée peut s'exprimer sous la forme :

$$\Gamma \quad = \quad K_p(x_d - x) + K_v(\dot{x}_d - \dot{x})$$

où :

- Γ représente la commande envoyée au robot à travers la ressource physique.
- K_v représente un gain en vitesse.
- K_p représente un gain en position.
- $\dot{x}_d$ représente la vitesse désirée.
- $\dot{x}$ représente la vitesse courante.

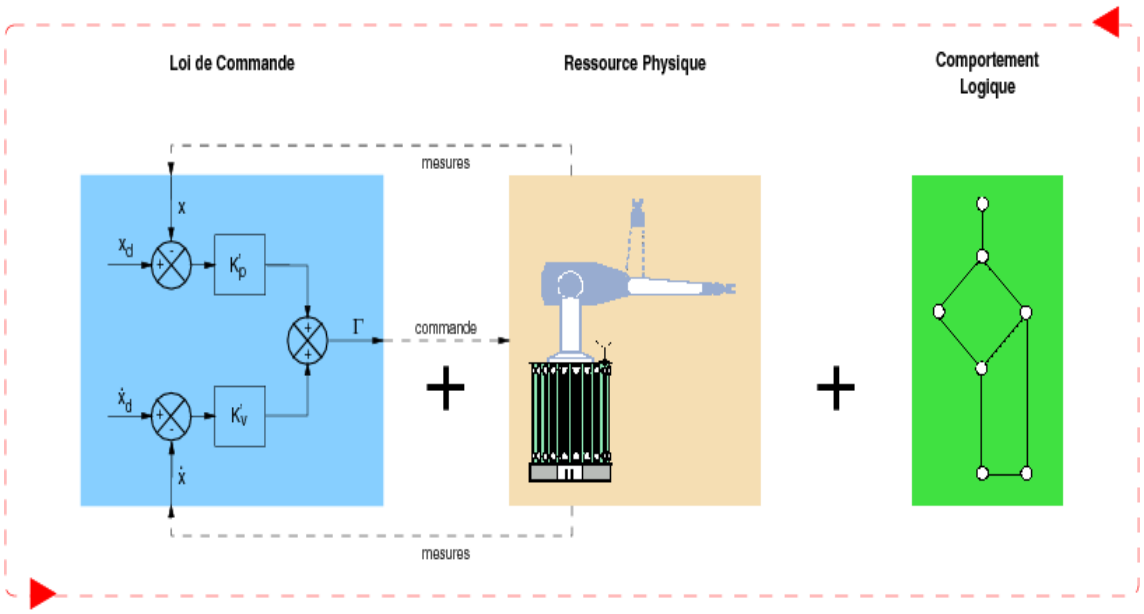


FIG. 5.2 – Définition de la Tâche Robot

– x_d représente la position désirée.
– x représente la position courante.

La loi de commande est évaluée de manière périodique suivant une cadence qui permet d'assurer la stabilité de la partie robotique. La période d'exécution de la loi de commande dépend des gains choisis pour l'asservissement et est déterminée le plus souvent de manière expérimentale.
La spécification de la loi de commande demande la décomposition en un enchaînement de *modules algorithmiques*. Ces modules implantent chacun une partie du calcul de la commande et ils communiquent entre eux à travers des ports de données.
La **réutilisation** des composants algorithmiques est favorisée par cette décomposition en modules : un même module peut être utilisé pour le calcul d'autres commandes et donc il sert à spécifier d'autres TÂCHES ROBOT, de plus, on isole clairement les différentes parties du calcul de la commande.

On complète la spécification de la loi de commande en précisant la période d'exécution de chaque module qui la compose.

La ressource physique. C'est la seule partie de la TÂCHE ROBOT qui enferme des informations **robot dépendantes**. En effet, elle centralise les appels au *pilote matériel* du robot. Elle constitue en fait l'interface entre la loi de commande de la TÂCHE ROBOT et le robot lui-même. La ressource physique permet d'envoyer des consignes aux actionneurs et fournit les informations proprioceptives et extéroceptives qui vont être utilisées par la loi de commande pour commander la partie du robot concernée.Cette entité est à la fois la source du flot de données continu qui va être transformé au travers de la loi de commande et le destinataire des consignes issus du calcul sur le flot de données.

Comportement Logique. Chaque TÂCHE ROBOT est assortie d'un comportement logique générique [Kapellos 94]. Il est défini en terme de séquence de signaux prédéfinis et caractéristiques de la gestion du contrôle de base de toute TÂCHE ROBOT. Ce comportement logique définit trois phases distinctes au sein d'une action : la phase d'initialisation, la phase de commande et la phase de terminaison. L'utilisateur peut paramétrer ce comportement générique pour gérer les spécificités d'une action par l'intermédiaire de trois types de signaux ou *événements* reçus ou émis dans chacune de ces phases :

i. les *préconditions* qui doivent être vérifiées avant que la TÂCHE ROBOT ne démarre ;
ii. les *exceptions* qui sont divisées en trois catégories :
   Type 1 : ces exceptions sont gérées par la tâche elle-même et signifient à la tâche courante qu'elle doit changer de mode de fonctionnement ;
   Type 2 : ces exceptions signifient à la tâche courante qu'elle doit arrêter son exécution et le contrôle est transféré au niveau supérieur, i.e. le niveau contrôle ;
   Type 3 : ces exceptions entraînent l'arrêt du système.
iii. les *postconditions* représentent les événements dont a besoin la tâche pour terminer.

Ces aspects discrets sont gérés par un formalisme propre aux systèmes réactifs : le **langage ESTEREL** [Berry 91],[Berry 96].
ESTEREL est un langage de programmation synchrone textuel, de style impératif destiné à la programmation des systèmes réactifs (consulter l'annexe B de ce document pour plus d'informations sur ces systèmes). Une des particularités importantes de ce langage est qu'il se base sur une sémantique mathématique. Cette caractéristique permet l'utilisation de méthodes formelles pour valider les programmes écrits avec ce langage.
Donc, le comportement logique de la TÂCHE ROBOT, que l'utilisateur peut particulariser avec des événements typés (préconditions, postconditions et exceptions) est transformé automatiquement en un programme ESTEREL.
Par conséquent, le comportement de la TÂCHE ROBOT peut être validé de manière formelle, du fait de l'utilisation de ESTEREL. Ainsi, cette gestion spécifique du contrôle contribue à assurer la **sûreté** des actions robotiques au niveau du contrôle.

5.1.2 Le niveau contrôle

Définition : Les entités manipulées par ce niveau sont les **Procédures Robot** (ou PR) [Kapellos 94], et sont définies comme suit :

> *Les* PROCÉDURES ROBOT *permettent de composer* **LOGIQUEMENT** *et de manière hiérarchique des* TÂCHES ROBOT *et d'autres* PROCÉDURES ROBOT *au sein de structures de complexité croissante, permettant de spécifier des actions mais aussi des applications complètes.*

Par conséquent, ce niveau ne manipule pas de données numériques (c'est le rôle du niveau fonctionnel), mais plutôt des événements discrets. La communication entre ces deux niveaux se fait par l'intermédiaire de signaux (i.e. les événements discrets). Chaque TÂCHE ROBOT est assortie d'une *vue abstraite* qui synthétise les caractéristiques principales de son comportement logique, de ses paramètres et de la ressource physique utilisée (par exemple, le nom du capteur). C'est cette *vue abstraite* qui constitue l'interface avec le niveau contrôle.

Une application ORCCAD est définie en terme de séquence, de mise en parallèle, de préemption d'un ensemble de TÂCHES ROBOT et de PROCÉDURES ROBOT nécessaires pour la mise en œuvre de la mission robotique.

Par exemple, une PROCÉDURE ROBOT peut s'exprimer comme :

```
faire (ACTION_1 puis ACTION_2) jusqu'à EVENEMENT_Y
puis
faire (ACTION_3 en parallèle ACTION_4) jusqu'à EVENEMENT_Z
```

Cette approche permet de rester indépendant du type de robot envisagé, car le contrôle est indépendant de l'implantation des actions élémentaires. Par conséquent, ceci permet d'assurer **la portabilité** de la spécification de l'application robotique : une même PROCÉDURE ROBOT pourra être utilisée sur un système robotique différent, si les TRs associées sont définies.

Spécification. La spécification sous forme de PROCÉDURES ROBOT s'exprime :
- par un programme ESTEREL dans les cas simples ;
- par le langage MAESTRO lorsque l'on veut spécifier des missions plus complexes.

Le langage de programmation de mission MAESTRO [Coste-Manière 97], [Turro 99]. Ce langage permet de composer des TÂCHES ROBOT, suivant une syntaxe propre au langage, pour spécifier des PROCÉDURES ROBOT afin d'élaborer finalement une mission robotique. MAESTRO se distingue par une sémantique opérationnelle, et par son interfaçage avec un éditeur de programme structuré, utilisant l'environnement CENTAUR [Borras 88], qui facilite la programmation de l'application. Ce langage est complètement dédié à la programmation d'applications robotiques et utilise des opérateurs simples et caractéristiques du domaine robotique pour spécifier rapidement le contrôle de l'application robotique.
L'interface entre les programmes MAESTRO et les actions définies sous ORCCAD est assurée par des fichiers de description, appelés *vues abstraites*. Ces vues abstraites contiennent l'information nécessaire au contrôle de chaque TR, i.e. son nom, le nom de la ressource physique commandée, le type des paramètres, les événements paramètres du comportement. Elles sont générées automatiquement par MAESTRO à partir des informations d'ORCCAD sur les TRs.

Une autre caractéristique importante de ce langage, est qu'un programme MAESTRO est traduit à l'exécution en un programme ESTEREL. Par conséquent, on peut utiliser les mêmes outils de vérification que pour un programme ESTEREL.

Vérification. ORCCAD permet des vérifications sur les programmes en ESTEREL. Ces vérifications consistent principalement à analyser l'automate issu du contrôle logique de l'application pour en valider des propriétés particulières.

Ces propriétés concernent :

- les caractéristiques structurelles des PRs, i.e. des caractéristiques génériques à toute application. On vérifie des propriétés telles que :
  - la *vivacité*, où l'on s'assure des possibilités de bonne terminaison ;
  - la *sûreté*, où l'on vérifie que les exceptions qui sont susceptibles d'être déclenchées durant l'application sont bien prises en compte ;
  - les *conflits*, si plusieurs ressources physiques sont utilisées dans l'application, on vérifie qu'à chaque moment on ne commande qu'une unique fois chaque ressource physique.
- l'interaction entre les TRs et les PRs qui composent l'application. Ici, l'automate global de l'application est réduit pour ne contenir que les informations logiques relatives aux appels de démarrage et de terminaison des TRs et des PRs.
- la cohérence du comportement avec les spécifications. On sélectionne un ensemble de TRs, de PRs et d'exceptions par rapport auquel l'automate global de l'application est observé.

Ces différentes techniques de vérification permettent d'assurer la **sûreté** de fonctionnement au niveau contrôle. Le comportement global de l'application est obtenu en combinant le comportement des TÂCHES ROBOT et des PROCÉDURES ROBOT qui composent l'application. Par conséquent, on peut valider au niveau logique toute l'application.

En plus des méthodes de vérification sur le langage ESTEREL, MAESTRO, quant à lui, permet de faire des vérification par *analyse statique*, automatiquement, sans intervention de l'utilisateur.
Ces méthodes d'analyse statique[1] servent à évaluer les propriétés des programmes sans les exécuter : par exemple, on peut vérifier automatiquement qu'une variable du programme MAESTRO est toujours affectée avant d'être lue, ou encore que le programme ne possède pas de boucle infinie, i.e. qu'il peut se terminer.

Le tableau 5.1.2 récapitule les entités de spécification définies dans la méthodologie ORCCAD.

5.2 ORCCAD : un environnement de programmation

La méthodologie décrite ci-dessus (au paragraphe 5.1) est mise en œuvre dans un environnement de programmation qui distingue strictement quatre étapes dans le cycle de programmation d'une application robotique, à savoir : la spécification, la vérification, la simulation et l'exécution. L'environnement se présente graphiquement comme l'illustre la figure 5.3.

Cet environnement a été bâti comme une boîte à outils extensible intégrant les outils dédiés à chaque aspect de la programmation d'une application robotique : depuis les outils pour spécifier les TRs et les PRs, en passant par la vérification de la partie comportementale et la simulation, jusqu'à la génération de code automatique correspondant à l'application

[1] [Lacan 98] présentent un cas intéressant d'analyse statique fait sur du code ADA issus des programmes du lanceur Ariane 5

Entités ORCCAD	Définitions	Propriétés
TÂCHE ROBOT ou TR	Action robotique élémentaire	
Ressource Physique	Actionneur autonome à commander	Non portabilité
Module	Entité algorithmique	Réutilisabilité
Automate	Entité du comportement logique	Vérification formelle (ESTEREL) Sûreté au niveau logique
PROCÉDURE ROBOT ou PR	Ensemble de TRs ou de PRs ordonné **de manière logique**	Portabilité Vérification formelle (ESTEREL) Sûreté comportementale
MAESTRO	Environnement de spécification, Implantation et de validation des PRs	Vérification formelle par analyse statique Vérification formelle du comportement logique par ESTEREL Sûreté comportementale

TAB. 5.1 – Récapitulatif des entités de spécification manipulées en ORCCAD

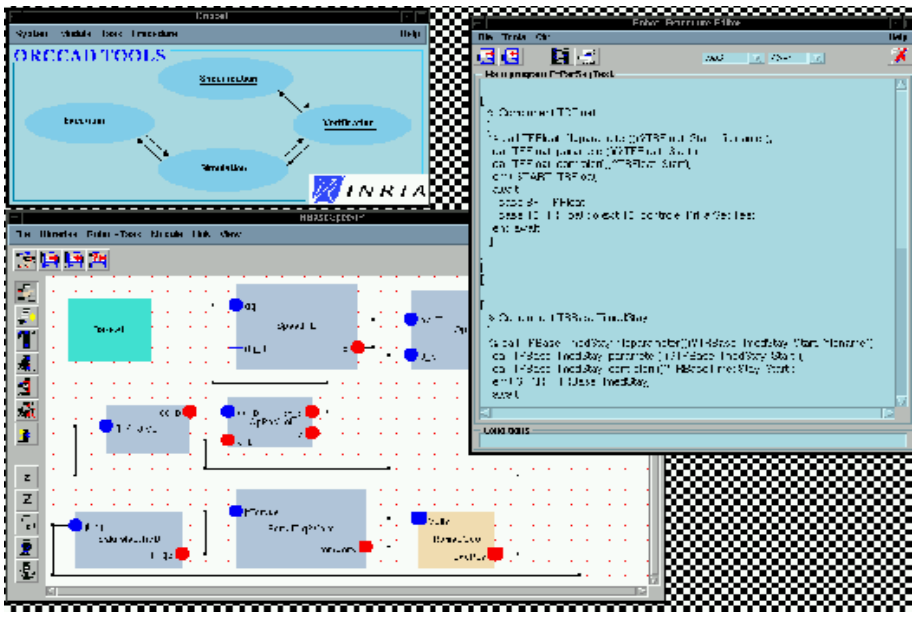


FIG. 5.3 – Une vue d'ensemble de l'environnement ORCCAD

(voir figure 5.4). A chacune de ces phases (spécification, vérification, simulation et exécution) correspond une interface graphique particulière intégrant les outils dédiés. Ces outils communiquent par des classes C++ constituant le noyau logiciel d'ORCCAD.

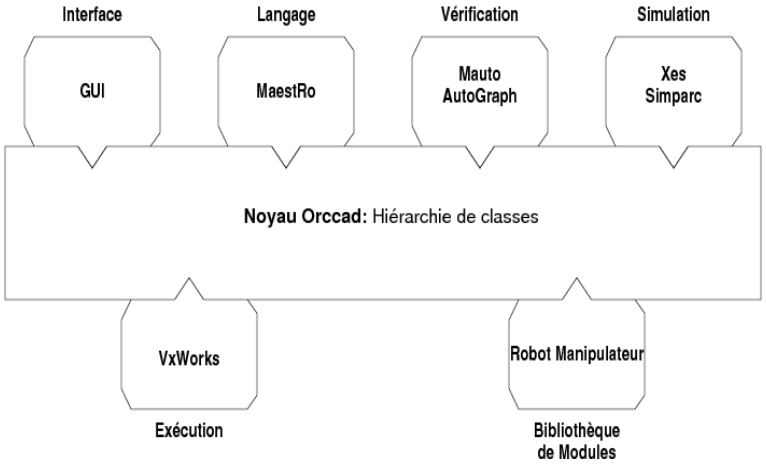


FIG. 5.4 – ORCCAD conçu comme une boîte à outils

5.2.1 Spécification

5.2.1.1 Les TÂCHES ROBOTS

Spécification sous forme de schémas-blocs. La spécification des actions élémentaires (TRs) se fait graphiquement, sous forme de *schémas-blocs*. Chaque composant de la TÂCHE ROBOT (ressource physique, module algorithmique, comportement logique) est identifié par un bloc particulier (code de couleur associé au type de bloc), avec des caractéristiques propres.
Ces blocs communiquent entre eux à travers des ports reliés par des liaisons. Ces ports véhiculent différents types de données en fonction des blocs reliés. On distingue :

- le *port de donnée*. Il désigne une donnée numérique. Il permet de relier les modules algorithmiques entre eux ou de fournir une donnée sur le *port driver* d'une ressource physique. Mais il permet aussi de relier le bloc comportement logique appelé module *automate* avec les modules algorithmiques, qui peuvent produire des événements.
- le *port driver*. Il désigne une donnée numérique. Il permet de relier la ressource physique aux modules algorithmiques soit pour fournir une donnée aux modules algorithmiques soit pour recevoir une donnée des modules algorithmiques.
- le *port événement*. Il désigne un événement discret. Il permet de relier les modules algorithmiques et le module automate, exclusivement.
- le *port paramètre*. C'est un port particulier car il ne peut être utilisé que sur des modules algorithmiques et il n'est relié à aucun autre port d'aucun bloc. Les paramètres peuvent être spécifiés :
  i. soit par une valeur numérique fixe ;

ii. soit par un identifiant qui est associé alors à un paramètre de la TÂCHE ROBOT et dont la valeur sera fixée par une autre TÂCHE ROBOT.

Ces ports sont utilisés comme *variables* lors de la spécification du code d'implantation des modules.

Interface de spécification des modules. Pour chaque bloc, l'utilisateur doit spécifier le nom du bloc, et les ports d'entrée et de sortie :

- dans le cas du module *automate*, l'utilisateur précise pour chaque port *événement* le type correspondant à la typologie de la méthodologie ORCCAD, i.e. précondition, post-condition, ou exception (T1, T2, T3).
- dans le cas des modules *ressource physique ou algorithmique*, les ports *driver, de donnée, et paramètre* sont identifiés par un nom fourni par l'utilisateur, un type (Integer, Float, Double), une nature (scalaire, vecteur, matrice).
  De plus, l'utilisateur complète la définition de ce type de module en fournissant le code[2] qui implante la fonctionnalité du bloc.
  Pour ce faire, le code est décomposé en différentes phases, qui correspondent aux étapes définies par le comportement logique de la TÂCHE ROBOT, i.e. initialisation, exécution et fin de l'action (voir le paragraphe 5.1.1).
  La décomposition du code diffère suivant que l'on traite une ressource physique ou un module algorithmique :

[2] ORCCAD supporte le langage C ou C++ pour le code des modules

Ressource physique	Module Algorithmique
inc.h : ce fichier contient la liste des déclarations externes et des fichiers à inclure.	
var.c : ce fichier contient la liste des variables locales, les ports d'entrée et de sorties des blocs étant exclus de cette liste.	
init.c : ce fichier décrit les traitements à faire pour initialiser la ressource physique.	init.c : ce fichier implante les traitements à faire comme initialisation du module à exécuter. Tous les fichiers init.c des blocs définis pour une action sont exécutés après que le signal de déclenchement de l'action a été reçu par l'automate.
reinit.c : ce fichier décrit les traitements à faire lorsqu'il est nécessaire, en cours d'exécution, de modifier certaines caractéristiques de la ressource physique.	compute.c : ce fichier décrit les calculs concernés par le module algorithmique. Dans ce fichier, on va utiliser les variables définies dans le fichier var.c mais on va manipuler en tant que variable du programme les ports spécifiés comme entrées et sorties du module.
end.c : ce fichier décrit le traitement correspondant à la terminaison du bloc. Les fichiers `end.c` spécifiés dans une TÂCHE ROBOT sont exécutés après que l'automate a reçu un signal de postcondition.	

Bibliothèque hiérarchique de modules. ORCCAD offre la possibilité de constituer une bibliothèque hiérarchique de modules algorithmiques réutilisables en fonction de leur fonctionnalités, par exemple, les modules servant pour la robotique de manipulation.
Les modules, regroupés dans une bibliothèque, sont entièrement définis (codes du module, ports de communication). L'utilisateur peut facilement les insérer graphiquement par *drag and drop* dans l'interface de spécification de TÂCHE ROBOT.

Spécification des contraintes temporelles. L'utilisateur achève la spécification de la TÂCHE ROBOT en spécifiant, pour chaque module algorithmique composant la TR, les contraintes temporelles associées, i.e. la fréquence d'échantillonage, priorité, etc. Pour ce faire, une interface graphique particulière est proposée.

5.2.1.2 Les PROCÉDURES ROBOTS

Les PROCÉDURES ROBOTS correspondent à un arrangement logique de TRs et de PRs. Comme on l'a vu au paragraphe 5.1.2, cet arrangement peut être fait soit en langage

ESTEREL, soit en langage MAESTRO.

Programmation en ESTEREL. L'environnement ORCCAD propose un éditeur particulier qui facilite la programmation de l'application en ESTEREL, mais sans toutefois éluder le besoin de connaître ce langage.
En effet, cette interface permet d'insérer automatiquement l'appel à une TR ou à une PR dans le programme. Pour ce faire on dispose de deux menus qui listent les TRs et PRs déjà spécifiées.

Cet appel encapsule déjà les instructions ESTEREL nécessaires pour lancer la TR ou la PR. En effet, chaque lancement d'action TR ou de PR implique un protocole d'interaction particulier avec le niveau commande. Ce protocole est lié au comportement générique propre à l'architecture ORCCAD et définit automatiquement pour chaque TR et PR, et qui distingue différentes phases telles que l'initialisation, le fonctionnement nominal et la terminaison.

Cependant, si on veut écrire un arrangement plus complexe de TRs ou de PRs, il faut utiliser les instructions du langage ESTEREL.
A titre indicatif, voici quelques instructions qui sont souvent utilisées pour composer une application :

- ***instr***$_i$ **;** ***instr***$_j$: la mise en séquence des instructions *instr*$_i$ et *instr*$_j$.
- ***instr***$_i$ $\|$ ***instr***$_j$: la mise en parallèle des instructions *instr*$_i$ et *instr*$_j$. L'instruction parallèle se termine lorsque les deux branches d'instructions *instr*$_i$ et *instr*$_j$ sont finies.
- **emit** ***Sig*** : émission du signal *Sig*.
- **await** ***Sig*** : attente de l'occurrence du signal *Sig*.
- **trap** ***Tag bloc_instructions*** **exit** ***Tag*** **end trap** : mécanisme de sortie de *bloc* lorsque `exit Tag` est exécuté. Cette structure permet notamment de gérer la préemption dans un bloc d'instructions.

Pour plus d'informations sur la syntaxe ESTEREL, on peut consulter l'annexe C ou se référer à [Berry 92].

L'éditeur permet aussi de définir des *événements externes* à l'application. En effet, l'application robotique peut nécessiter des événements qui ne sont pas produits par les TRs ou PRs qui la composent. Ce sont ces *événements externes* et ils peuvent correspondre par exemple à des signaux issus d'une interface utilisateur servant à la supervision de l'application.

Une fois l'application définie, l'interface génère automatiquement le programme ESTEREL qui fusionnent tous les composants présents (comportements des TRs et PRs) dans le programme de l'application. Ce programme représente le comportement global de l'application.

Programmation en MAESTRO. MAESTRO [Turro 99] est un langage spécifique : ses opérateurs sont dédiés au contrôle, et les entités élémentaires manipulées sont des actions robotiques ou des événements.

Ce langage comprend les trois composants suivants :

– Un *éditeur structuré* de PROCÉDURES ROBOT, qui permet la saisie, aussi confortablement que possible, d'applications. Cet éditeur se base sur l'environnement CENTAUR [Borras 88].
– Un *interpréteur de vues abstraites* des TÂCHES ROBOT. Les actions mises à la disposition de l'utilisateur sont connues de l'environnement MAESTRO au travers de fichiers de description, les *vues abstraites*.
– Un *traducteur* du langage MAESTRO vers le langage ESTEREL. Ce traducteur implante un certain nombre d'analyses statiques sur les programmes, que nous décriront par la suite. Le programme ESTEREL produit sera incorporé à ceux générés par ORCCAD pour obtenir le contrôleur global de l'application.

Voici la syntaxe adoptée par MAESTRO pour définir une vue abstraite, cette syntaxe est décrite en notation BNF (Backus-Naur Form)[Backus 60] :

```
Vue_Abstraite   : := TASK nom :
                     PARAM : typeparam_s
                     RETURN : returnparam
                     ROBOT : robot
                     precond_s
                     postcond_s
                     exceptionT2_s
                     exceptionT3_s
nom             : := chaine de caractères
typeparam_s     : := void | typeparam | typeparam_s,typeparam
typeparam       : := INT | STRING | BOOL | FLOAT |
                     type défini par l'utilisateur
returnparam     : := void | typeparam
robot           : := nom
precond_s       : := void | precond | precond_s precond
precond         : := PRECOND nom
postcond_s      : := void | postcond | postcond_s postcond
postcond        : := POSTCOND nom
exceptionT2_s   : := void | exceptionT2 | exceptionT2_s exceptionT2
exceptionT2     : := T2 nom
exceptionT3_s   : := void | exceptionT3 | exceptionT3_s exceptionT3
exceptionT3     : := T3 nom
```

On remarque que la vue abstraite ne fait pas apparaître les exceptions de type 1. En effet, ces exceptions sont exclusivement gérées localement au niveau de la TÂCHE ROBOT. Elles n'ont pas besoin de figurer dans les informations formant l'interface avec le niveau supérieur, i.e. les PROCÉDURES ROBOT.

Les différents opérateurs du langage sont :

✦ le lancement d'une action robotique élémentaire :

START *<nom de l'action>* **(** *<paramètres>* **)**

Cette instruction est bloquante tant que l'action élémentaire lancée n'est pas terminée. En particulier, si l'action concernée n'a pas de postconditions, elle ne peut pas terminer d'elle même, il faudra donc veiller dans ce cas à encapsuler la primitive **START** d'un opérateur de préemption.
Si l'action robotique élémentaire doit retourner une valeur (ce qui doit être précisé dans la vue abstraite de l'action), l'instruction devient :

START *<nom de l'action>***(***<paramètres>***) : ***<variable>*

- l'appel d'une procédure, au sein d'une autre :

  PROCEDURE *<nom de procédure>*

- La séquence d'instruction : elle permet d'exécuter une suite d'instructions les unes après les autres.

  SEQ(*<liste d'instructions>* **)**

  La séquence se termine lorsque la dernière instruction de la liste finit.

- la préemption :

  DO *<instruction>* **UNTIL** *<nom d'événement>*

  Un **DO ... UNTIL** peut terminer soit parce que son corps termine, ou bien parce que l'événement indiqué est reçu. Si le corps du **DO ... UNTIL** contient plusieurs branches parallèles d'exécution, elles sont toutes stoppées simultanément.
  Si, en réaction à l'événement, et seulement dans ce cas, on veut exécuter une instruction, on utilise l'instruction suivante :

  DO *<instruction>* **UNTIL** *<liste de réactions>*

  avec les réactions de la forme :

  <nom d'événement> **->** *<instruction>*

  Il est important de signaler ici que cet opérateur est le seul qui permet de spécifier une exécution conditionnelle. En effet, MAESTRO ne possède pas de mécanisme d'évaluation de "condition sur l'environnement" ni de variables donc pas d'opérateur de choix. Toutes les alternatives d'un programme sont choisies réactivement sur occurrence d'un événement.

- la répétition infinie d'instructions

  LOOP(*<instruction>* **)**

- la concurrence (ou parallélisme)

  PAR(*<liste d'instructions>* **)**

  Cet opérateur permet d'exécuter simultanément plusieurs actions robotiques sur des ressources physiques différentes.
  Cette instruction termine quand toutes les branches d'exécution de la liste d'instructions sont terminées.

Pour plus d'informations sur la syntaxe de ce langage, on peut consulter [Turro 99].

On construit le programme de l'application en arrangeant les actions avec les opérateurs. Grâce à l'éditeur structuré, l'élaboration du programme se fait progressivement par clics de souris sur la liste des opérateurs du langage, sur la liste des actions disponibles, etc.

Une fois le programme de l'application spécifié, le programme obtenu est transformé, de façon transparente à l'utilisateur, en un automate de contrôle. En effet, comme nous l'avons dit précédemment en début de ce paragraphe, le programme MAESTRO est traduit en un programme ESTEREL. Par conséquent, un programme MAESTRO bénéficie des mêmes outils

de vérification que les programmes ESTEREL, ce qui va permettre de le valider indirectement (sur la traduction ESTEREL du programme).

Discussion. Nous avons décrit les deux manières selon lesquelles on peut spécifier les PROCÉDURES ROBOT. Il est intéressant de comparer ces deux approches de programmation :

- ESTEREL est plus expressif en cela qu'il dispose de plus d'opérateurs de manipulation de signaux. De plus, il a été conçu pour la programmation parallèle et réactive. Il permet donc de prendre en compte les besoins de synchronisation particulier des applications robotiques. Cependant, en fonction de la complexité de l'application à spécifier, cela devient vite très difficile à utiliser. D'une part, il suppose la manipulation de **signaux**, qui sont les éléments de base du langage, ce qui nécessite un apprentissage important.
  D'autre part, ESTEREL repose sur *l'hypothèse de synchronisme parfait*, i.e. la diffusion des signaux est faite de manière instantanée. Ceci introduit des problèmes nouveaux et qui peuvent être déconcertant pour le programmeur robotique, lors de la conception des programmes. En particulier, les problèmes de *causalité* dans les programmes, phénomène qui se produit lorsque l'on émet un signal dans le même instant où on le reçoit, peuvent être contraignant à résoudre.
  Ceci est plus particulièrement vrai lorsqu'il s'agit de corriger des erreurs inévitables du programme. En effet, le programme écrit au niveau de la PROCÉDURE ROBOT ne représente qu'une petite partie du contrôleur global de l'application ORCCAD, qui est compilé par ESTEREL : le contrôle des TRs est généré automatiquement par ORCCAD. Par conséquent, le programmeur de l'application n'a qu'une visibilité limitée et aucune aide pour corriger les bogues dans la spécification des PRs.
- MAESTRO, quant à lui, est un langage plus abstrait que ESTEREL. Mais,il a été spécialement conçu pour l'élaboration d'applications robotiques. En particulier, ses opérateurs sont spécifiques à la manipulation d'entités robotiques.
  En fait, MAESTRO a extrait de ESTEREL les caractéristiques nécessaires pour répondre aux besoins de spécification d'une mission robotique en terme d'arrangement logique d'actions robotiques.
  Il rend les subtilités du langage ESTEREL transparentes à l'utilisateur, car il adopte des hypothèses simplificatrices et il assure la traduction correcte et automatique en ESTEREL.
  Ses opérateurs sont proches du langage naturel, et sont plus simples et plus faciles à appréhender pour un novice en système réactif.

Au vue de ces considérations, il apparaît que MAESTRO est plus approprié pour programmer de manière efficace et simple les PROCÉDURES ROBOT.

5.2.2 Vérification

L'étape de vérification concerne la validation du comportement logique des TRs et plus généralement de l'application. Elle permet d'analyser les relations temporelles, au sens logique, entre les actions et les événements. Par exemple, on pourra vérifier qu'une action est toujours lancée après qu'un événement particulier soit émis.

Elle se base sur l'analyse de l'automate fini issu de la compilation de la partie contrôle de l'application qui est exprimé en ESTEREL. Cet automate incorpore à la fois la partie comportement propre au niveau fonctionnel, i.e. spécifique à chacune des TÂCHES ROBOT, et la partie comportement propre au niveau application, i.e. spécifique aux PROCÉDURES ROBOT qu'elles aient été définies directement en ESTEREL ou en MAESTRO, le tout dans une même structure cohérente. On pourra utiliser cet automate pour vérifier le comportement global de l'application ou prouver que certaines propriétés sont satisfaites.

L'environnement ORCCAD propose une interface particulière vers des outils de vérification comportementale de la partie contrôle :

- **FC2MIN, FC2SYMBMIN** [Bouali 96].
  Ces outils permettent de :
  – minimiser l'automate produit ;
  – d'observer l'automate à travers des programmes auxiliaires (les *observateurs*) ;
  – d'occulter une partie des signaux ;
  – et de valider l'équivalence entre automates.
  La minimisation de l'automate permet de faciliter l'observation de l'automate représentant le comportement de l'application, car on réduit le nombre d'états à inspecter et on limite l'explosion combinatoire du nombre d'états de l'automate. De plus, il est possible de réduire l'automate en fonction d'une propriété particulière que l'on cherche à valider. Ceci se traduit par un parcours dans l'automate guidé par les signaux relatifs à la propriété à vérifier.
  Les *observateurs* sont des programmes mis en parallèle du programme ESTEREL principal et qui interagissent avec lui. Ces observateurs servent à valider ou invalider certains comportements de l'automate en lui envoyant des signaux, et en testant les signaux provenant du programme principal. L'observateur effectue des diagnostics sur le succès ou l'échec de l'automate à valider un comportement particulier.
  L'occultation des signaux permet de ne sélectionner que les signaux utiles et que l'on souhaite analyser dans l'automate.
- **ATG** [Roy 90].
  Il s'agit d'un éditeur graphique d'automate qui permet de visualiser globalement ou partiellement les états et les transitions de l'automate résultant du programme ESTEREL issu de la partie comportementale.

Au niveau de leur utilisation au sein de ORCCAD :

- FC2SYMBMIN, FC2MIN sont utilisés de manière transparente à l'utilisateur lors de la compilation du code ESTEREL traduisant le comportement global de l'application.
  Ainsi, l'automate qui est visualisé par ATG correspond à un automate minimisé.
- Le système ORCCAD combine ces outils, FC2SYMBMIN, FC2MIN et ATG, de façon prédéfinie pour proposer à l'utilisateur des automates réduits équivalents à l'automate global, mais utilisables pour vérifier **visuellement** certaines propriétés du comportement logique de l'application. Ces propriétés peuvent concerner le séquencement des TRs, le déclenchement de certaines actions en fonction de signaux particuliers, ou encore la prise en compte des signaux d'urgence tels les exceptions de type 3.
  ORCCAD ne permet pas automatiquement de générer des *observateurs* sur des propriété-

tés particulières pour valider systématiquement une des caractéristiques du comportement global de l'application. L'utilisation faite des outils FC2SYMBMIN, FC2MIN par ORCCAD est limitée à la minimisation de l'automate global pour permettre une visualisation sous ATG plus facile.

Discussion. Analysons l'usage du langage MAESTRO et ESTEREL sous le point du vue de la vérification.
MAESTRO intègre un traducteur en ESTEREL, qui transforme un programme de MAESTRO → ESTEREL (puis par compilation du programme ESTEREL → C). Donc, ces programmes peuvent bénéficier des mêmes outils de vérification au niveau comportemental qu'un programme écrit directement en ESTEREL.
Cependant, ceci implique que l'on suppose que le traducteur MAESTRO → ESTEREL est correct, i.e. valide. Dans le cas contraire, on risque d'obtenir des programmes ESTEREL erronés suite à une traduction fausse, et ils seraient inutilisables pour les vérifications.

Mais, comme nous l'avons déjà dit plus haut (au paragraphe 5.1.2), MAESTRO propose un autre type de vérification sur ses programmes. Il utilise des méthodes d'analyse statique qui permettent d'assurer automatiquement que le programme respecte *la sémantique opérationnelle* du langage.
Ces *méthodes d'analyse statique* consistent à prédire statiquement et automatiquement les comportements possibles des programmes, sans les exécuter. On interprète le programme avec des critères très abstraits, classes d'instructions par classes d'instructions. Ces critères concernent par exemple l'affectation effective d'une variable, l'intervalle de définition des indices d'un tableau, le typage des données. Cette double facette du langage MAESTRO au niveau de la validité des programmes accroît l'intérêt de cet outil.

5.2.3 Simulation

L'environnement ORCCAD considère deux types de simulation :

i. la simulation sur la partie logique de l'application, i.e. le contrôle de l'application.
   Elle est réalisée avec l'outil XES [FC2TOOL WEB Page].

ii. la simulation hybride, i.e. ce qui concerne à la fois le contrôle et les commandes en tenant compte d'une modélisation des contraintes liées à la partie matérielle (hardware) du système à commander.
   Elle est réalisée avec l'outil SIMPARC

5.2.3.1 Simulation du contrôle avec XES

XES [FC2TOOL WEB Page] est un simulateur graphique de programme ESTEREL qui permet de jouer des séquences d'exécution de manière interactive. On peut exécuter de manière interactive la partie relative au contrôle correspondant à l'application globale. Pour ce faire, l'utilisateur va envoyer des signaux (sous forme de clics souris) parmi les événements d'entrée possibles au programme ESTEREL gérant le contrôle de l'application. Ces signaux comprennent en particulier les préconditions, exceptions et postconditions avec lesquelles le

comportement des TRs a pu être paramétré.
En réaction aux signaux transmis, l'utilisateur peut vérifier si les événements produits en sortie correspondent aux lancements d'actions, par exemple.

5.2.3.2 Simulation avec SIMPARC

SIMPARC [Astraudo 92] est un outil qui permet de simuler à la fois la partie dynamique du robot, le contrôle et les caractéristiques matérielles du système, tels que les fréquences d'échantillonnage, les délais de communication.

La simulation avec SIMPARC requiert la modélisation préalable du robot à commander sous forme d'équations différentielles, des capteurs (ultra-son, sonar) sous forme de modèles à base de lancer de rayons, l'environnement 3D du robot sous forme de polygônes, et enfin la description de l'architecture du système informatique, sous forme de composants matériels, décrits dans une bibliothèque spéciale qui est fournie par l'outil SIMPARC.

La partie correspondant au contrôle de l'application générée par ORCCAD n'est pas soumise à modélisation. Elle est intégrée à la partie modélisée lors de la génération du code exécutable de simulation.

En conséquence, SIMPARC permet de simuler le système robotique au complet : les aspects continus, les aspects discrets et les aspects de la plateforme d'exécution temps réel.

5.2.4 Exécution

Une fois que l'on a spécifié, vérifié et simulé les spécifications de l'application, on peut passer à l'étape d'exécution.
Cette étape consiste à traduire la spécification de l'application en un code C++ temps réel suivant une plateforme d'exécution donnée (par exemple, VXWORKS, SUNSOLARIS, etc.).

Pour ce faire, ORCCAD gère cette traduction en deux parties :

i. la génération de code indépendant de la plateforme d'exécution, à savoir :
   - la partie fonctionnelle de l'application, i.e. les codes des modules implantant les TRs ;
   - la partie contrôle implantée en ESTEREL, qui est transformée en un programme C automatiquement par le compilateur ESTEREL ;

ii. l'intégration de ce code avec les routines de la plateforme d'exécution.
   Le code de l'application globale résultant est composé :
   - d'un ensemble de tâches temps réel périodiques et qui sont dédiées aux aspects continus (calcul des lois de commande principalement) ;
   - de l'automate de contrôle généré en C ;
   - et d'une interface entre ces deux entités, pour gérer la communication dans un sens (contrôle → tâche) ou dans l'autre (tâche → contrôle).

Une interface particulière est proposée à l'utilisateur avec laquelle il va sélectionner la plateforme d'exécution cible et les informations nécessaires à la génération de l'exécutable (par exemple, bibliothèques pour l'édition de lien).

L'utilisateur dispose également d'une interface particulière qui va lui permettre de sélectionner les données (numériques ou événements) produites au cours de l'application et qu'il souhaite examiner en cours d'exécution. Ces données sont très utiles si l'on veut analyser, à

l'exécution, la dynamique et le comportement de l'application.
Pour ce faire, ORCCAD met à la disposition de l'utilisateur un certain nombre de possibilités : visualiser les données dans la sortie standard de la plateforme, dans un fichier, ou à travers une fonction. Ces données doivent être sélectionnées avant la génération du code temps réel.

5.2.5 Comparaison TÂCHE ROBOT vs TÂCHE VISION

Dans le paragraphe 4.2.3, nous avons fourni une définition de la TÂCHE DE VISION (ou TV) comme :

une chaîne périodique de ***transformations paramétrées*** *de l'information visuelle fournie par un* ***capteur****. Cette chaîne de transformations est contrôlée au niveau logique pour gérer le respect des contraintes temporelles et assurer la conformité du résultat de la transformation.*

Si on compare cette définition avec celle de la TÂCHE ROBOT que nous avons présentée au paragraphe 5.1.1, on peut remarquer que :

- ces deux entités s'articulent autour d'une boucle de traitement temps réel périodique.
- la TR distingue rigoureusement la partie de calcul qui transforme un flot de données de manière continue, la partie flot de contrôle, et la ressource commandée.
- on peut voir les transformations successives opérées sur le flot de données continu issu de la caméra (image, liste chaînée des contours, etc.) comme la *loi de commande* de la TV. De la même manière, on peut faire se correspondre le comportement logique de la TR et le flot de données discret qui fournit des informations sur la cohérence du flot de données continu.
- un point très important dans la TV, est la gestion des paramètres et en particulier l'adaptation des paramètres. C'est un point important où la TÂCHE VISION et la TÂCHE ROBOT différent, même si cette dernière peut gérer des changements de mode de fonctionnement.

Face à ces analogies, il paraît intéressant d'utiliser l'environnement ORCCAD, qui est bâti autour du formalisme de TÂCHE ROBOT, pour mettre en œuvre nos TÂCHES DE VISION et plus généralement nos applications de vision.

Outre le formalisme de TR, l'architecture ORCCAD présente d'autres points forts pour la mise en œuvre des applications qui nous intéressent.

- La hiérarchisation de la mise en œuvre d'une application : on distingue strictement (i) un niveau dans lequel on va faire des calculs, que l'on contrôle localement, le niveau fonctionnel, et (ii) un niveau de contrôle global dans lequel on va manipuler les entités qui s'occupent des calculs indépendamment de leur implantation, à savoir de manière logique, le niveau contrôle.
- La formalisation du contrôle en terme de *systèmes réactifs*, ce qui permet de faire de vérifications et des validations formelles de la partie contrôle logique de l'exécution de l'application.
- La gestion stricte et transparente des contraintes temporelles, avec la génération automatique du code exécutable sur la plateforme robotique cible.
- La mise à disposition d'outils dédiés pour les différents niveaux de la hiérarchie : le

contrôle est géré par le langage MAESTRO ou ESTEREL, le niveau fonctionnel est géré sous forme de schémas-bloc avec une gestion implicite du contrôle en ESTEREL.

5.3 Conclusion

Dans ce chapitre, nous avons présenté l'architecture ORCCAD qui est conçue pour la spécification, la validation, la simulation et l'implantation d'applications robotiques.
Nous avons décrit la méthodologie et l'environnement associé au système ORCCAD.

Les points forts de cette approche reposent sur :

- la hiérarchisation de la mise en œuvre d'une application. ORCCAD distingue strictement deux niveaux de spécification :
  - le niveau *fonctionnel* qui manipule les TÂCHES ROBOT. Ces TRs sont des boucles de traitements temps réel le plus généralement périodiques. Ces tâches distinguent séparement la partie calcul (i.e. loi de commande), la gestion de la ressource robotique commandée et un contrôle local ;
  - le niveau *contrôle* qui manipule les PROCÉDURES ROBOT. Ces PRs correspondent à l'ordonnancement logique des actions définies au niveau fonctionnel, i.e. les TRs.
- la TR peut changer de mode de fonctionnement en cours d'exécution.
- la formalisation du contrôle logique de l'application en terme de *systèmes réactifs*, et en particulier avec le langage ESTEREL. Cette formalisation permet de valider rigoureusement la partie logique de l'application.
- la gestion stricte et transparente des contraintes temporelles des traitements.
- des outils dédiés pour faciliter la spécification de l'application sur chaque niveau de la hiérarchie. Le niveau contrôle propose le langage MAESTRO, le niveau fonctionnel propose une spécification sous forme graphique à base de schémas-blocs. L'exécutif est automatiquement généré en intégrant les aspects temps réel en fonction de la plateforme cible sur laquelle va être exécutée l'application.

En outre, nous avons vu que la notion de TÂCHE ROBOT définie par ORCCAD est très proche de la notion de TÂCHE DE VISION que nous avons décrite au paragraphe 4.2.3.

Par conséquent, dans le chapitre 6, nous allons détailler comment nous avons utilisé ce système pour mettre en œuvre l'application de vision temps réel présentée dans l'introduction de ce mémoire (chapitre 2), à savoir la "mamy-sitter". Cette expérience d'implantation avec ORCCAD va nous servir, par la suite (chapitre 7), pour analyser l'adéquation d'ORCCAD à spécifier les applications de vision temps réel. Cette analyse nous conduira à présenter les adaptations que nous avons mises en œuvre pour qu'ORCCAD réponde mieux aux prérogatives des applications de vision temps réel.

Chapitre 6

Première mise en œuvre de la "Mamy-Sitter" avec ORCCAD/MAESTRO

Dans le chapitre 5, nous avons présenté l'architecture ORCCAD et la méthodologie de développement associée. Dans ce chapitre, nous allons voir en détail comment nous avons mis en œuvre avec le système ORCCAD l'application de la "mamy-sitter", décrite au chapitre 2, lors de la présentation du contexte de ce travail.

Dans ce contexte, les actions élémentaires considérées ne sont pas les TÂCHES ROBOT (implantation d'une loi de commande robotique), mais des TÂCHES VISION qui implantent une chaîne de transformation d'image en temps réel. Nous avons vu dans le paragraphe 5.2.5 que ces deux entités ont une structure commune mais diffèrent par le type de traitements et par la gestion des paramètres. Par conséquent, dans ce chapitre nous allons présenter l'utilisation du système ORCCAD en tant qu'environnement de spécification de TÂCHE VISION, plutôt que de TÂCHE ROBOT.

6.1 Présentation générale de l'application

Dans ce chapitre, nous allons nous intéresser à une application de vision temps réel pour laquelle on dispose d'une caméra que l'on peut commander pour modifier les positions en *pan*.
Cette caméra va suivre de manière passive (sans mouvement de la caméra) une cible selon un champ de vue restreint. Dès que la cible sort du champ de vue, on effectue une phase de suivi actif où la caméra va se déplacer en fonction des mouvements de la cible. Cette action s'achève lorsqu'il n'y a plus de cible à suivre. Alors la caméra retourne à sa position initiale et on reprend l'activité de détection et de suivi passif.
A tout moment, un superviseur peut passer le dispositif en mode manuel, i.e. arrêter l'action en cours, et effectuer des manœuvres de téléopération sur la caméra.

Cette application peut être envisagée comme une version simplifiée de ce que pourrait être l'application de la "mamy-sitter", présentée au paragraphe 2.5.2. La simplification - toute relative - s'explique par le fait que notre travail s'est surtout porté sur les aspects temps

réel des applications de vision. Aussi, l'interprétation de scène présentée au paragraphe 2.5.2 n'est pas envisagée ici.

Lorsque l'on veut mettre en œuvre une telle application avec ORCCAD, il faut repérer les boucles de traitements périodiques définissant un mode de fonctionnement particulier.

En conséquence, notre application est caractérisée par quatre boucles de traitements périodiques :

i. la détection et le suivi passif de cible dans la scène ;
ii. le suivi de cible actif ;
iii. la téléopération sur la caméra ;
iv. la commande de positionnement de la caméra.

Les boucles de traitement (i) et (ii) doivent suivre la cadence de la dynamique de l'environnement à savoir au mieux la fréquence vidéo pour la première et la dynamique de la cible pour la seconde, sous peine de perdre la cible. Le processus de téléopération et la commande de positionnement de la caméra eux suivent la dynamique imposée par le système robotique à savoir la caméra.

La figure 6.1 schématise la décomposition en tâches de l'application que nous souhaitons réaliser.

6.2 Spécification des TÂCHES VISION

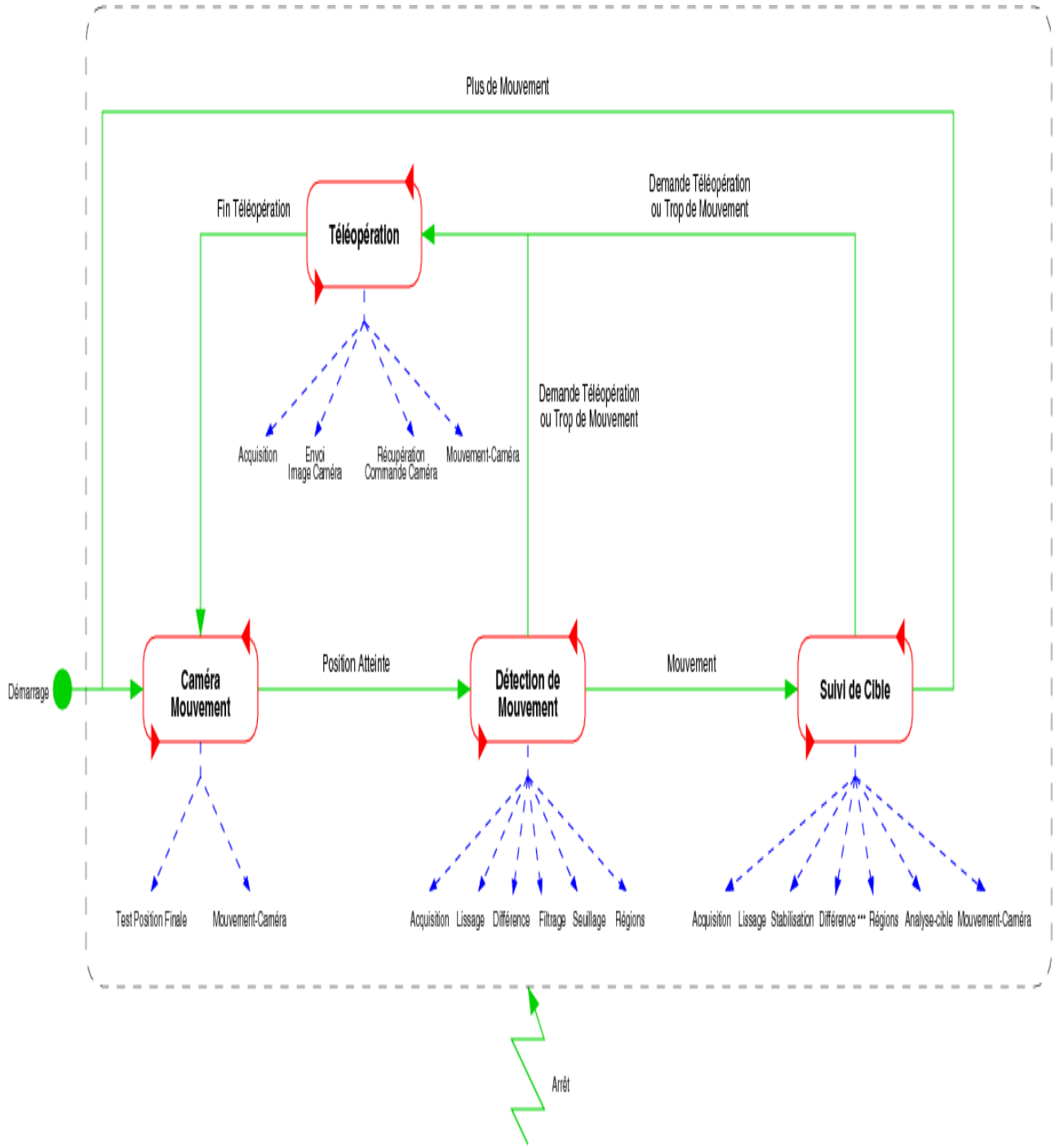
Cette application met en œuvre deux comportements temps réel différents que l'on implémente sous la forme de TÂCHE VISION : une qui effectue la détection de mouvement, l'autre qui s'intéressera au suivi d'objets.
On suppose que l'on dispose d'un dispositif de supervision qui permet de démarrer ou d'arrêter ces actions en cours d'exécution.

Maintenant que l'on a identifié les actions à spécifier en tant que TÂCHE VISION, on va devoir définir la ressource physique à commander, les modules algorithmiques et le comportement logique associé en terme de préconditions, postconditions et d'exceptions.

Pour ce qui concerne les modules algorithmiques, nous utilisons directement le résultat de l'analyse que nous avons faite au paragraphe 4.1.3, sur le suivi de régions en mouvement dans une scène d'intérieur.

L'identification des événements qui participent au contrôle logique des différentes TÂCHES VISION est plus délicate. Elle nécessite de se pencher sur la classification des événements d'une TÂCHE VISION définie par la méthodologie ORCCAD, à savoir *préconditions*, *postconditions exceptions de type 1, type 2 et type 3* et de déterminer en conséquence le comportement adéquat pour une TÂCHE VISION donnée.
Il faut savoir ce qui peut conditionner le démarrage de la TÂCHE VISION (*précondition*), et sa bonne terminaison (*postcondition*). Il faut déterminer les reparamétrages à gérer (*exception de type 1*), les dysfonctionnements auxquels est soumis la TÂCHE VISION et leur gravité (*exception de type 2* ou *exception de type 3*). Ensuite, il faut déterminer les modules qui vont être affectés par ce contrôle, à savoir, les modules algorithmiques qui produisent les événements et ceux qui sont concernés par la reparamétrisation.

Le tableau 6.1 fournit pour chaque TÂCHE VISION que nous avons identifiée les informations nécessaires à leur spécification.



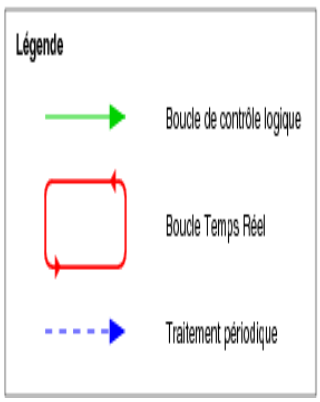


FIG. 6.1 – Décomposition de l'application

Une fois que l'on a décidé des caractéristiques de chaque action, on va commencer la mise en œuvre de l'application par la spécification et la validation des TÂCHES VISION, et remplir petit à petit une bibliothèque de TÂCHES VISION [1].

Pour ce faire, on utilise l'éditeur graphique de TÂCHE ROBOT d'ORCCAD, illustré par la figure 6.2.

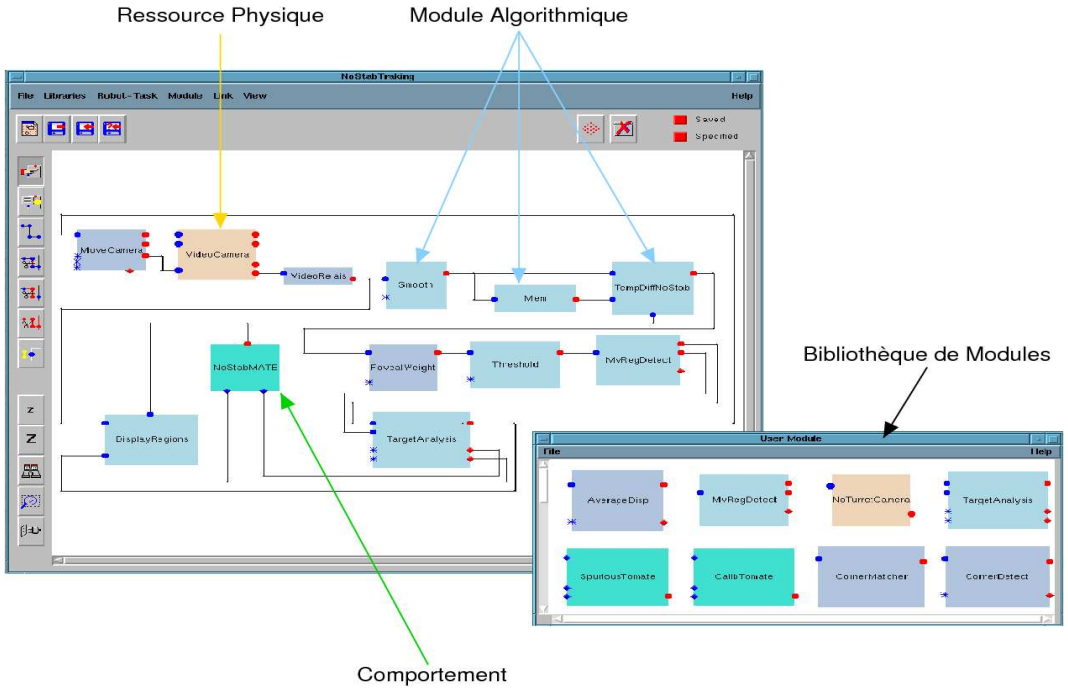


FIG. 6.2 – Spécification d'une Tâche Vision

Nous allons prendre comme exemple illustratif la TÂCHE VISION de suivi passif. Pour chacun de ces types de modules, nous présentons en image les caractéristiques de leur spécification en ORCCAD.

Le module ressource physique. Il modélise l'interface avec un système robotique commandé, dans notre cas il s'agit de la caméra commandée. Il est défini comme l'illustre la figure 6.3.

Le port de sortie *Video* fournit l'image acquise tandis que le port d'entrée *ChangePan* sert à transmettre la commande de changement de position en "pan" de la caméra.

On pourra remarquer que l'image fournie par la ressource physique est donnée sous la forme d'un entier, qui représente un pointeur sur l'image. Ceci s'explique par le fait qu'ORCCAD ne sait pas gérer des types autres que Integer, Float, Double.

[1] Au départ, la bibliothèque est vide.

Tâche Vision	Ressource Physique	Algorithmique	Comportement
Suivi passif	Caméra immobile Acquisition	lissage - mémorisation - différence - filtrage - seuillage - détection de cible - analyse de cible	**Postcondition :** mouvement hors champ demande utilisateur **Exception :** reparamétrisation caméra hors service
Suivi actif	Caméra mobile Acquisition Déplacement en pan	lissage - mémorisation - stabilisation - différence - filtrage - seuillage - détection de cible - analyse de cible - déplacement de caméra	**Précondition :** mouvement à suivre **Postcondition :** plus de mouvement demande utilisateur **Exception :** reparamétrisation déplacement erroné caméra hors service
Téléopération sur la caméra	Caméra mobile Acquisition Modification de zoom et focus Déplacement en pan	Visualisation d'image Calcul du déplacement en pan Calcul de la commande de zoom et focus	**Précondition :** demande utilisateur **Postcondition :** demande utilisateur **Exception :** Caméra hors service
Commandes de la caméra	Caméra mobile Déplacement en pan	Calcul du déplacement de la caméra	**Postcondition :** Fin de déplacement **Exception :** Caméra hors service

TAB. 6.1 – Spécification des actions sous forme de TÂCHE VISION

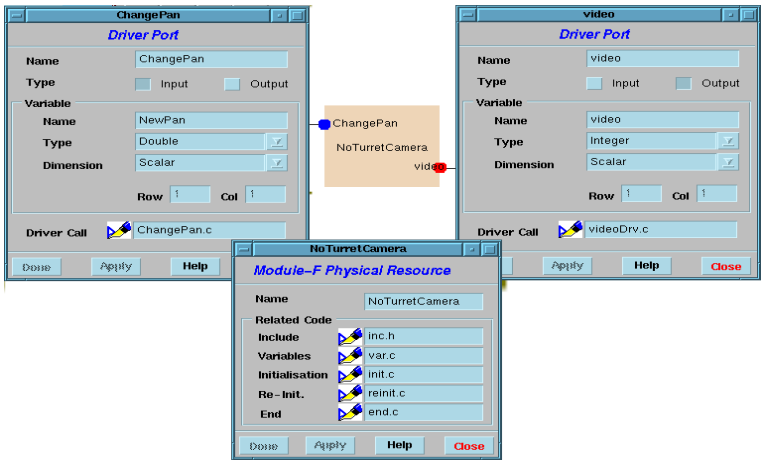


FIG. 6.3 – Spécification d'une ressource physique

Le module algorithmique. Il correspond à un traitement particulier, le plus souvent réutilisable, de la boucle périodique de calcul de la TÂCHE VISION. Le module, dont la spécification est illustrée par les figures 6.4, 6.5, 6.6, correspond à l'opération d'analyse des régions en mouvement détectées dans l'image.
La figure 6.4 présente la description des ports de données, événements et paramètres de ce module.
Arrêtons-nous sur la spécification des paramètres qui est un type de port exclusif au module algorithmique. Dans notre exemple, le module est caractérisé par deux paramètres de nature différente :

- *I_Thres* qui désigne un paramètre seuil d'intensité, dont la valeur est fixée à 5. Cette valeur est effective pendant la durée de la TÂCHE VISION.
- *Surf_Thres*, qui désigne le paramètre seuil de surface de région de mouvement. Il est associé au paramètre *Surf_Thres* de la TÂCHE VISION. Ceci veut dire que la valeur de ce paramètre peut évoluer en cours d'exécution de la TÂCHE VISION. De plus, ceci indique que lors du lancement de la TÂCHE VISION, la valeur du paramètre correspondant à *Surf_Thres* doit être spécifiée.

La figure 6.5 présente l'interface de définition des contraintes temporelles du module algorithmique. Cette interface est propre au module algorithmique. On y spécifie la durée du traitement à l'exécution, la durée valable lors de la simulation. Il est prévu de préciser le type de CPU sur lequel le module de traitement va être traité mais pour l'instant ORCCAD ne gère que des traitements sur un seul processeur. Par conséquent, la durée du traitement qui est spécifiée correspond à la cadence de l'application.

La figure 6.6 présente le code associé à ce module algorithmique. Les programmes illustrés correspondent au code C des fichiers var.c, init.c et compute.c. On note que chacun des ports d'entrée et de sortie est utilisé dans les programmes comme des variables que l'on affecte (port de sortie), ou encore qui servent pour le calcul (port d'entrée).

FIG. 6.4 – Spécification des ports d'un module algorithmique

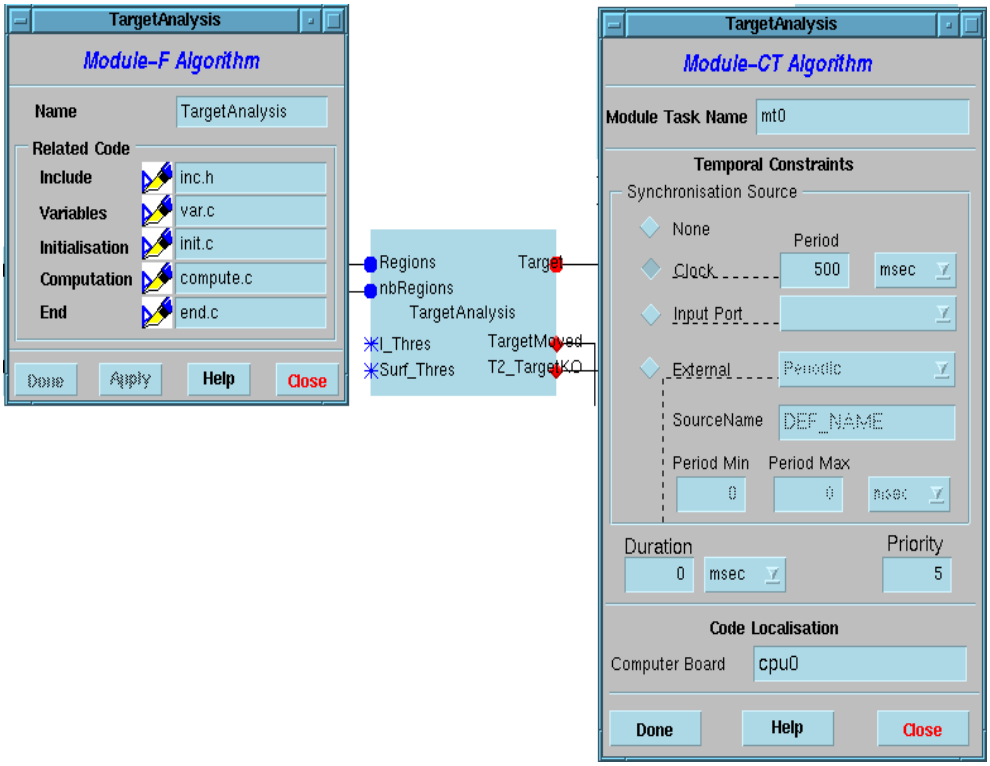


FIG. 6.5 – Spécification du module algorithmique

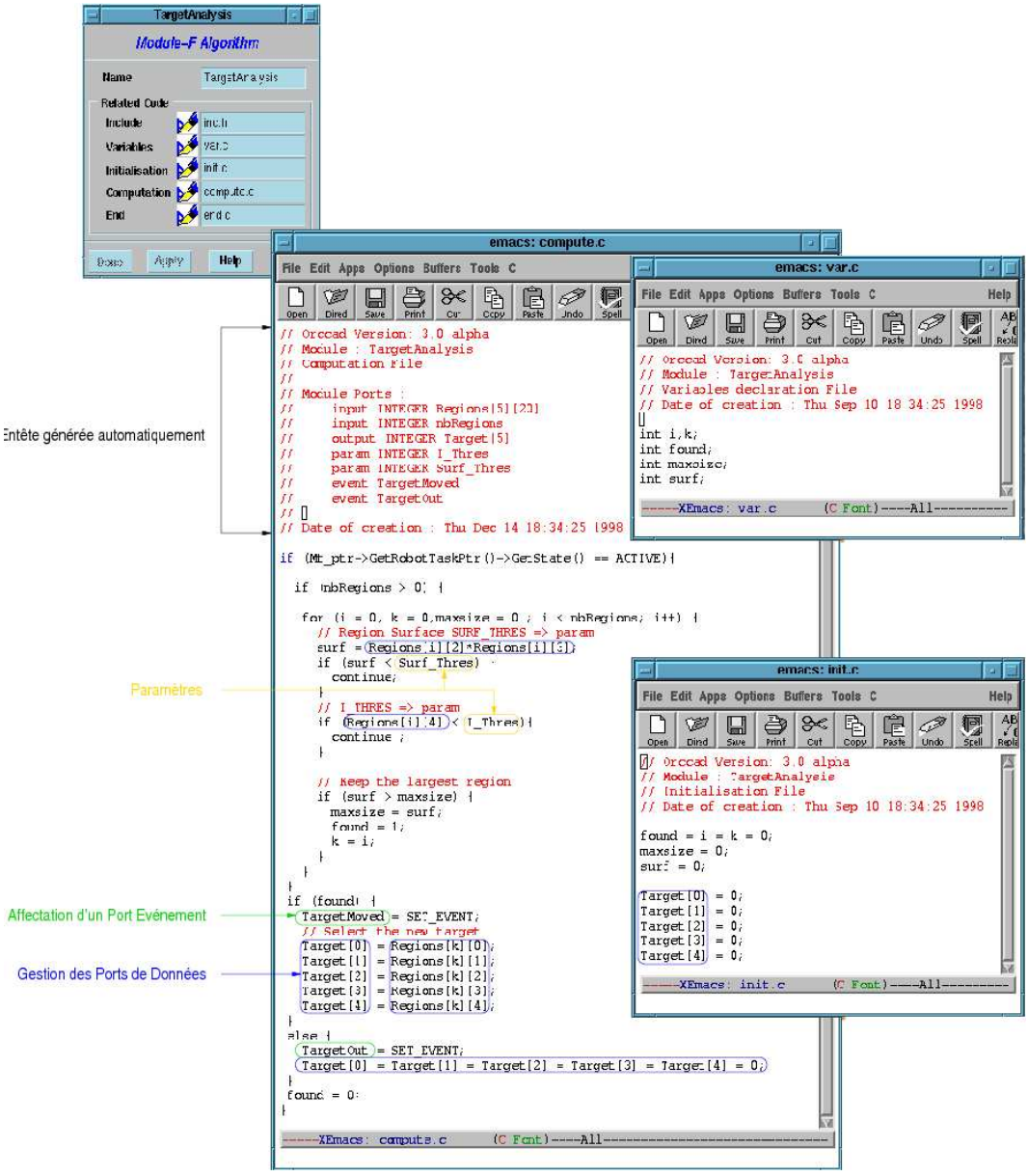


FIG. 6.6 – Spécification du code associé à un module algorithmique

Le module automate. Il décrit le comportement de l'action et est spécifié comme l'illustre la figure 6.7. C'est au moment de la définition de ce module que l'on va donner le typage désiré aux événements qui paramètrent le comportement de l'action.
Dans l'exemple, le comportement est paramétré par une exception de type 1 et par une post-condition.
On remarque la gestion particulière qui est à faire au niveau de l'automate de l'exception de type 1 : un port de donnée *MoveOK* en sortie est associé à l'événement *NoMove*, qui est une exception de type 1. Le port *MoveOK* est en fait un booléen qui est mis à vrai lorsque l'exception de type 1 associée, ici *NoMove*, survient pour notifier le besoin d'une reparamétrisation au niveau module algorithmique. Ce port de donnée communique avec les modules algorithmiques affectés par la reparamétrisation. Ainsi les modules algorithmiques, en fonction de la valeur de *MoveOK*, savent s'il faut ou non effectuer une reparamétrisation.

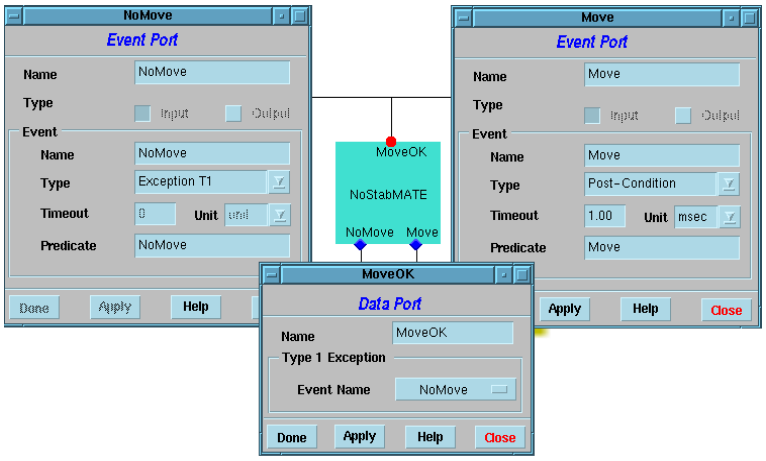


FIG. 6.7 – Spécification du module de comportement ou automate

Une fois chacun des modules spécifiés, on peut composer la TÂCHE VISION en ajoutant un à un les modules définis dans la bibliothèque de module puis en les reliant les uns aux autres à travers les ports (voir la figure 6.2).

6.3 Spécification de l'application sous forme de PROCÉDURE ROBOT

Comme nous l'avons dit au paragraphe 5.1.2, les PROCÉDURES ROBOTS peuvent être spécifiées soit par un programme ESTEREL, soit par un programme MAESTRO.

Nous allons présenter dans les paragraphes suivant comment on utilise ces deux approches pour spécifier les PROCÉDURES ROBOT et plus généralement une application.

6.3.1 En ESTEREL

L'environnement ORCCAD fournit directement un éditeur particulier dédié à la programmation en ESTEREL pour spécifier une application.

Cet éditeur propose un menu déroulant particulier qui permet de sélectionner les différentes TÂCHES VISION et Procédures Robot qui vont composer l'application. En conséquence, les vues abstraites de chacune de entités choisies apparaissent.
Ensuite, l'utilisateur les ordonnance en utilisant les instructions ESTEREL tels que la séquence ;, la mise en parallèle ||, la boucle loop .. end loop ou encore la gestion d'exceptions trap ... exit ... .

L'utilisateur peut spécifier des conditions externes qui représentent des signaux qui proviennent soit d'autres procédures robots, soit de l'environnement (par exemple, interface de supervision).

La figure 6.8 présente la spécification en ESTEREL de l'application quelque peu simplifiée par rapport à ce qui a été défini en introduction de cette section 6.1.
Cette application est composée des deux TÂCHES VISION de suivi passif et actif mis en séquence. Elles peuvent être préemptées par un signal provenant de l'utilisateur *User_Stop*.

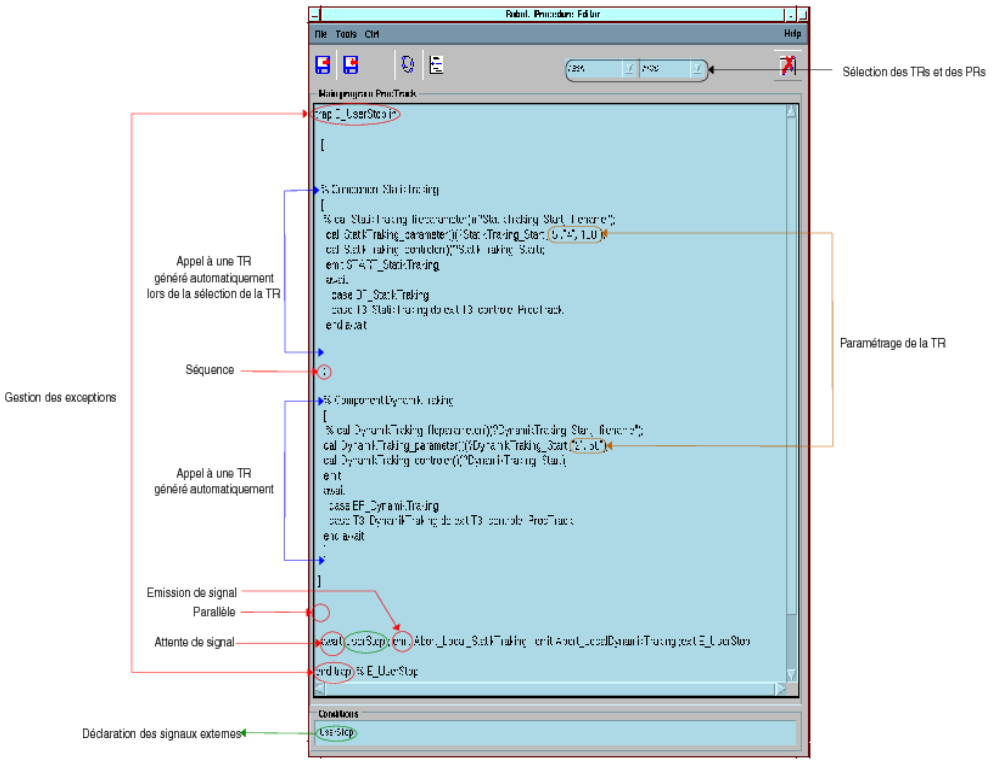


FIG. 6.8 – Spécification d'une application

Une fois l'application définie, l'interface permet de générer automatiquement le code ESTEREL relatif au contrôle de toute l'application. Ensuite, on pourra utiliser les outils de vérification (ATG, XES) sur ce programme.

Ici, l'application est suffisamment simple pour se passer de MAESTRO. L'application ne

gère qu'une seule ressource physique, donc cela ne requiert pas de gérer du parallélisme entre actions comme c'est le cas lorsque l'on doit gérer plusieurs ressources physiques à la fois. Dans ce cas, écrire le programme en ESTEREL devient vite très complexe, d'où l'intérêt de MAESTRO.

6.3.2 Avec le langage MAESTRO

Le langage MAESTRO est interfacé avec un autre outil, l'éditeur structuré CENTAUR, sur lequel ce langage de mission a été développé.

La mise en œuvre de l'application se fait par clic souris sur des menus contenant les différentes instructions du langage.

MAESTRO permet d'importer les vues abstraites de toutes les TÂCHES VISION définies par ORCCAD. Ainsi, l'utilisateur ajoute dans le programme par simple clic souris la TÂCHE VISION qui l'intéresse parmi celles qui ont été importées, et il n'a plus ensuite qu'à remplir les paramètres d'appel de la TÂCHE VISION.

La figure 6.9 présente la spécification en langage MAESTRO de l'application complète définie en introduction de cette section 6.1.

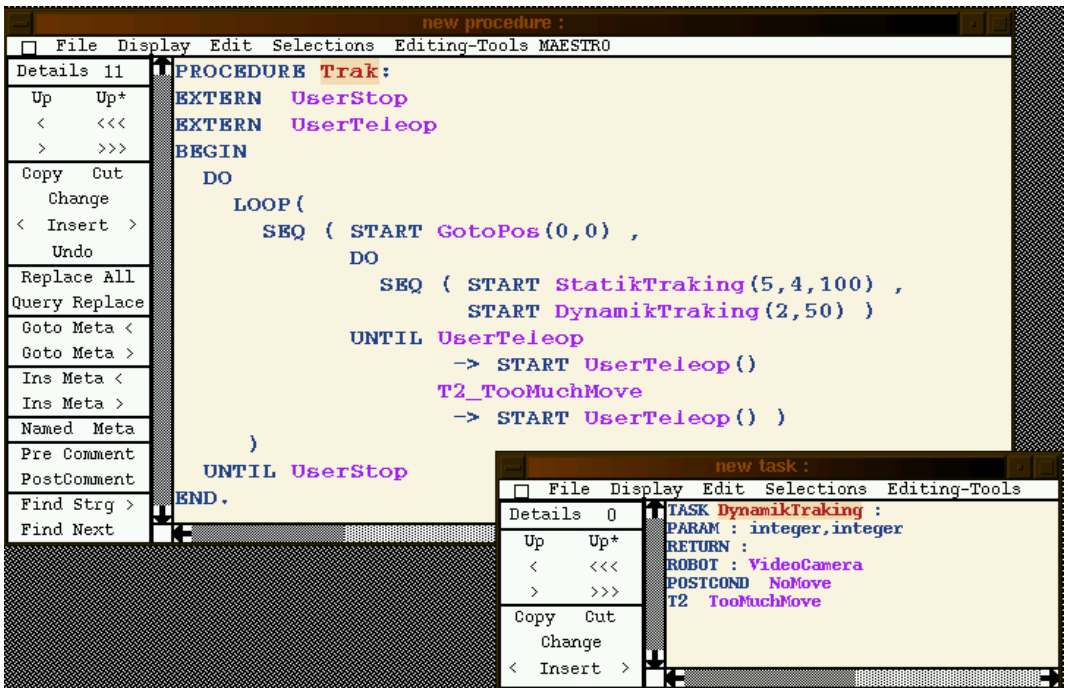


FIG. 6.9 – Programme MaestRo correspondant à l'application. On a fait apparaître pour une des TÂCHES VISION, *DynamikTraking*, la vue abstraite associée générée par MAESTRO.

6.4 Vérification et Simulation

La vérification peut se faire à plusieurs niveaux : soit au niveau de chaque TÂCHE VISION, soit au niveau de chaque PROCÉDURE ROBOT ou plus généralement au niveau de l'application globale.

Quelque soit le niveau considéré (le niveau TÂCHE VISION ou PROCÉDURE ROBOT), on peut décider de valider logiquement le contrôle, soit en visualisant l'automate représentatif du comportement avec l'outil ATG, et à ce moment là, on peut restreindre l'automate pour étudier certaines propriétés particulières (comme le montrent les figures 6.10 et 6.11), soit en simulant le comportement avec Xes (comme le montre la figure 6.12).

L'automate global de l'application définie par le programme MAESTRO présenté au paragraphe 6.3.2 est constitué de 52 **états** et 1976 **transitions**.

ORCCAD propose une interface graphique qui permet d'éditer l'automate de contrôle logique de l'application selon différents points de vue en fonction des propriétés que l'on souhaite vérifier. La figure 6.10 montre le panneau d'édition et l'automate généré pour ne voir du contrôleur logique que la partie relative au lancement des différentes TÂCHES VISION composant l'application.

Remarques sur la lecture de l'automate :

- les signaux d'entrée sont préfixés par ?
- les signaux de sortie sont préfixés par !.
- le label `tau` sur les transitions : il est utilisé pour indiquer qu'une transition existe entre deux états mais que les signaux correspondant ne concernent pas l'observation désirée par l'utilisateur.

La figure 6.11 montre le panneau d'édition et l'automate généré lorsque l'on veut observer l'automate au niveau des TÂCHES VISION : *StatikTraking*, *DynamikTraking* et *GotoPos*. Cet automate permet de vérifier, par exemple, que *DynamikTraking* est toujours suivi de *GotoPos* avant que *StatikTraking* ne soit lancé.

La figure 6.12 représente l'interface de simulation XES, avec laquelle on va pouvoir "jouer" par clics souris le programme ESTEREL traduisant le comportement logique d'une TÂCHE VISION, ici en l'occurrence, il s'agit de la tâche de suivi passif de mouvement.
L'utilisateur fait évoluer le programme en cliquant sur les événements d'entrée compris par le comportement et marqués en bleu. En conséquence, il voit comment le code réagit en étudiant les signaux de sortie produits notifiés en rouge, et où en est le programme par observation de la zone active mis en bleu.

Pour ce qui est de la simulation plus complète de l'application, nous n'avons pas utilisé le simulateur hybride SIMPARC car il ne permet pas d'intégrer facilement une modélisation des images produites par un capteur tel qu'une caméra.
Cependant les algorithmes mis en jeu dans les modules algorithmiques ont été préalablement testés sur des jeux d'images test.

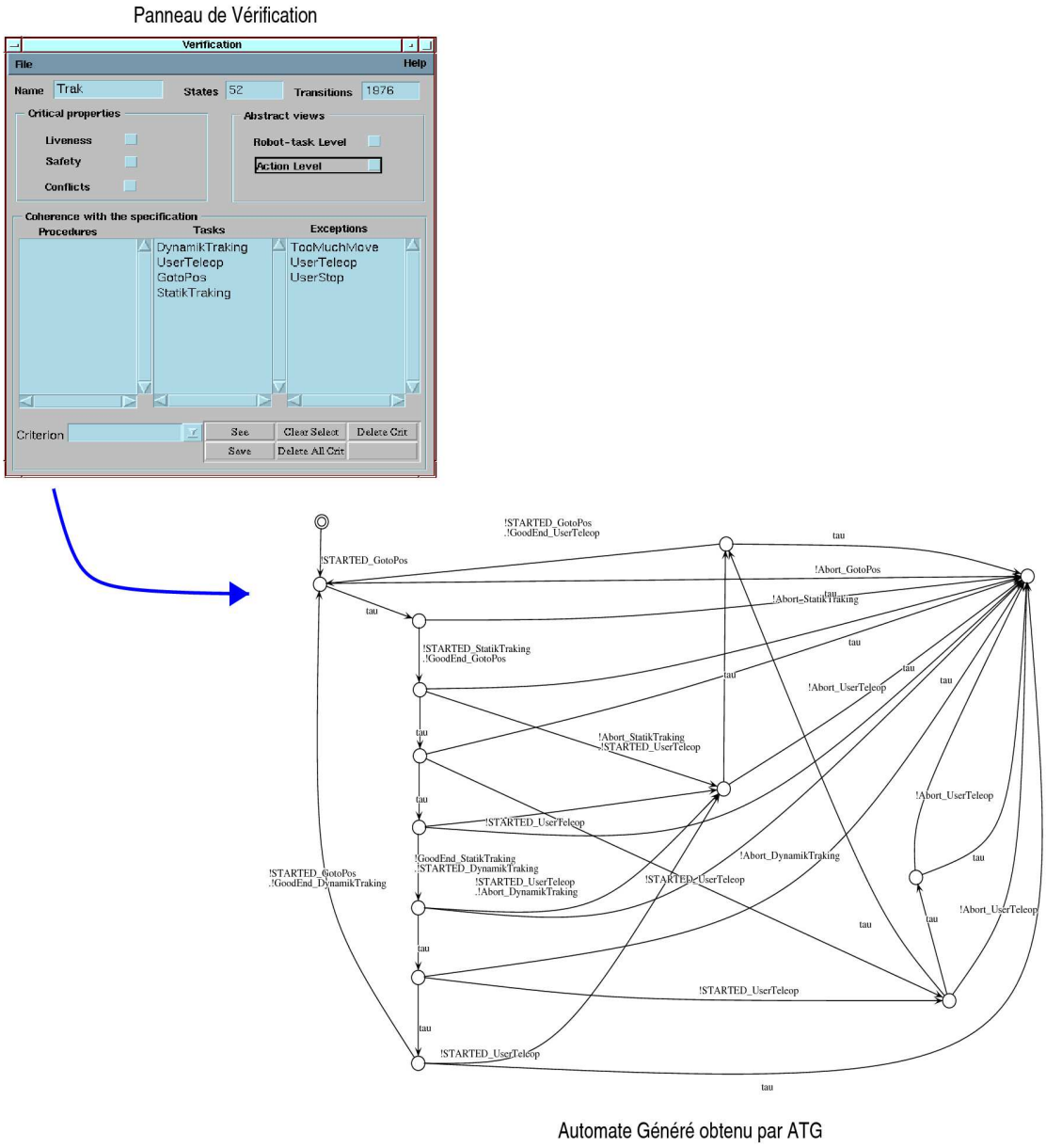


FIG. 6.10 – Visualisation sous ATG de l'automate, représentatif de l'application complète, restreint afin d'étudier le contrôle logique des actions utilisées. En l'occurrence, on observe l'enchaînement des événements de déclenchement et d'arrêt des différentes tâches composant l'application.

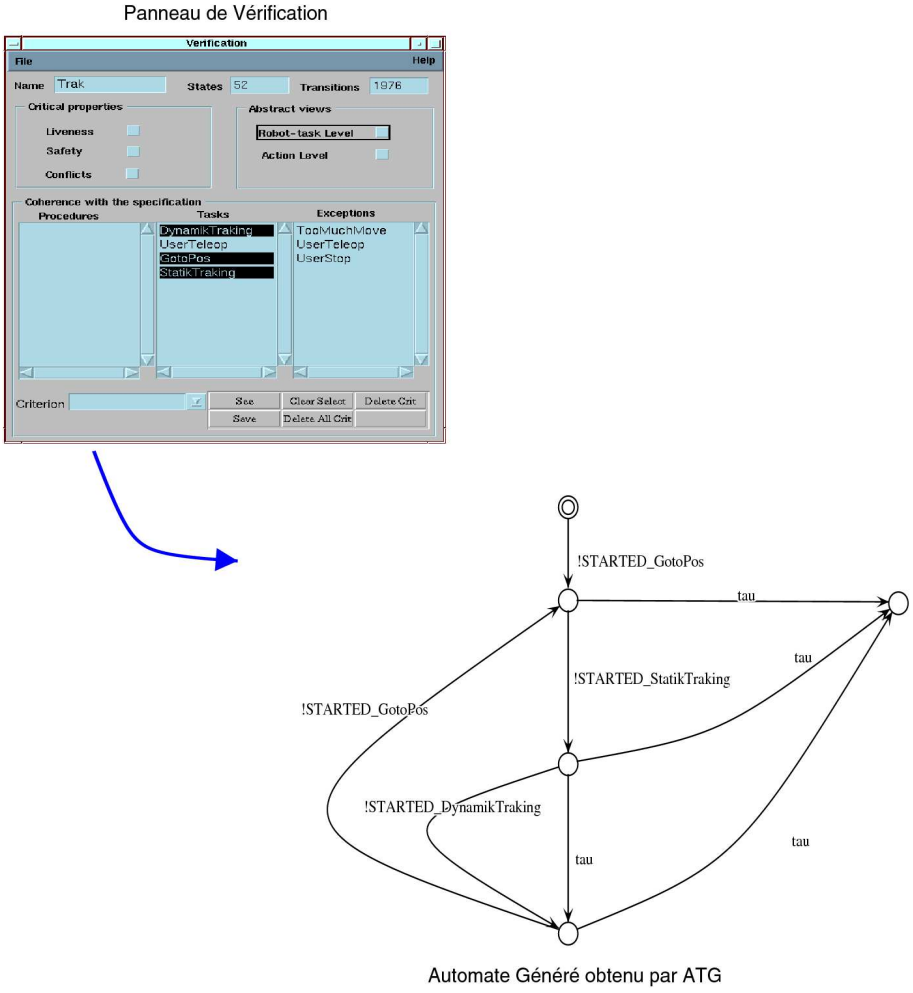


FIG. 6.11 – Automate réduit pour permettre la validation visuelle d'une propriété particulière du comportement logique d'une partie de l'application.

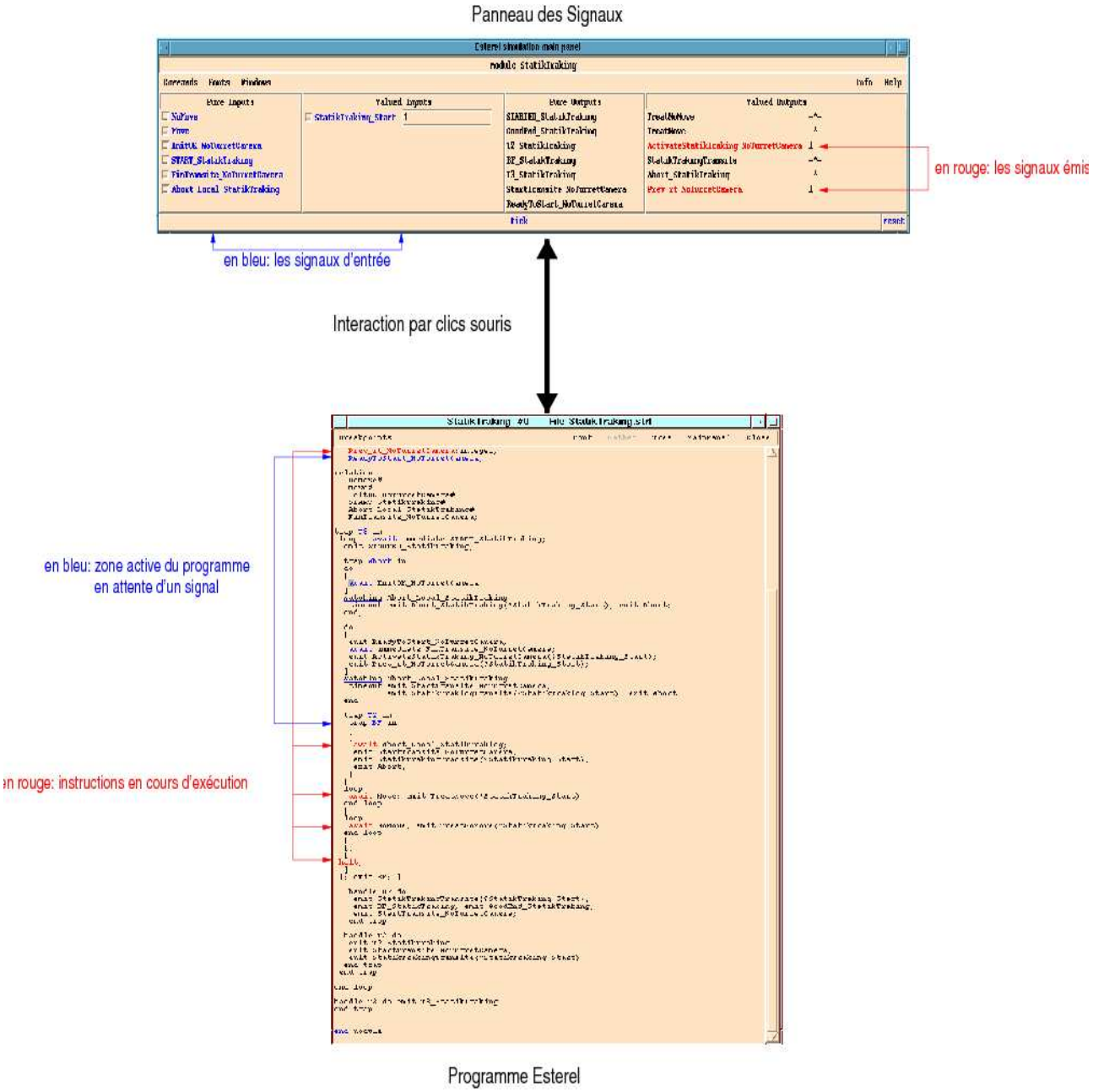


FIG. 6.12 – Interface de simulation de la partie comportement logique avec l'outil XES. Le panneau supérieur présente les signaux d'entrée (à gauche), de sortie (à droite) qui peuvent être respectivement envoyés au, ou produits par le programme ESTEREL qui figure dans la fenêtre inférieure. On sélectionne les signaux d'entrée à envoyer dans le panneau supérieur, on simule "l'instant" d'occurrence des signaux en cliquant sur le bouton *tick*. Alors, les signaux sont envoyés au programme et instantanément on visualise les signaux de sortie produits en réaction (panneau supérieur) et l'état du programme (fenêtre inférieure).

6.5 Génération de l'exécutable temps réel

Maintenant que la spécification logique a été validée, on peut générer l'exécutable.
Pour ce faire, on sélectionne l'éditeur graphique correspondant. Il est composé principalement de deux panneaux : un qui gère la génération de code et la génération du code exécutable, un autre gère les données que l'on veut tracer (facilité de débogage). Le troisième panneau sert à gérer le monitoring de l'application et est surtout utile avec le système VXWORKS. Dans le cadre de nos applications, nous n'en avons pas eu besoin.
Sur le panneau de génération de code et d'exécutable (voir la figure 6.13), on va choisir le système cible (VXWORKS, SUNSOLARIS, SIMPARC) qui détermine le type de code temps réel à générer, on donne un nom à l'exécutable, et on fournit la liste des bibliothèques de programmes nécessaires. Cette spécification rassemble les informations que l'on peut collecter dans une makefile UNIX, lorsque l'on compile un programme.

FIG. 6.13 – Panneau pour générer et compiler l'application

Avec le panneau de trace de données (voir figure 6.14), on sélectionne les données que l'on souhaite voir tracées et comment on souhaite visualiser ces données : à travers un fichier, la sortie standard, une fonction, ou encore l'interface WINDVIEW, propre au système VXWORKS. Nous avons choisi pour notre application d'avoir la trace sur la sortie standard, pour avoir un retour sur la dynamique des données en même temps que l'exécution de l'application.

Une fois toutes ces informations fournies, on retourne sur le panneau de génération de code, et on clique sur les boutons de génération de code, et de génération de l'exécutable.
La génération de l'exécutable consiste en la compilation et l'édition de liens du code généré. Des retours arrières peuvent être nécessaires si la génération de l'exécutable ne peut pas se faire suite à des erreurs de compilation, ou d'édition de lien. Alors il faut aller modifier le code des modules algorithmiques, et revenir sur l'interface de génération de code et d'exécutable.

FIG. 6.14 – Panneau pour tracer des données et des événements de l'exécutable généré par ORCCAD

6.6 Expérimentations

Cette application a été effectivement mise en œuvre en ORCCAD.

Au niveau du matériel informatique, nous avons travaillé sur le système de caméra monoculaire ARGÈS [ARGES 99]. C'est une plateforme expérimentale développée dans le projet ROBOTVIS pour tester les algorithmes de vision active 3D.

Le système ARGÈS est composé des éléments suivants :

- un caméra couleur CCD Acom1 PAL, contrôlée par ordinateur. Ses caractéristiques sont les suivantes :
  - une focale variable allant de $f = 5.9$ à $f = 47.2$, permettant de faire du zoom ;
  - un auto-focus numérique codé sur 10 bits ;
  - une interface vidéo et une interface RS232C ;
  - un rapport $S/N > 46dB$;
  - une résolution supérieure à $450x400$.
- une tourelle Pan-Tilt de chez ROBOTSOFT qui utilise une interface RS232C pour piloter le moteur.
- un degré de liberté linéaire de CHARLYROBOT, avec une résolution de 0.1 mm.
- un rail de $85cm$ sur lequel peut se déplacer la tourelle Pan-Tilt.
- un panneau d'acquisition SUNVIDEO basé sur la bibliothèque d'appels XIL.

La figure 6.15 présente ce robot-caméra.

Le système est piloté à travers un driver logiciel défini dans le fichier d'interface. On peut donc commander la caméra, modifier le pan, le tilt, la position de la tourelle sur le rail, acquérir des images que l'on pourra ensuite traiter.

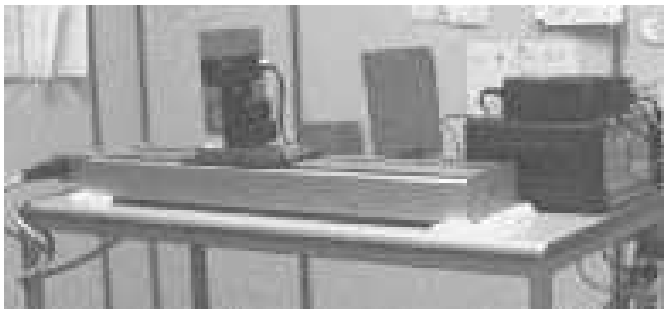


FIG. 6.15 – Le Système Argès

Ce système est relié par l'interface RS232C à une station de travail de type SUN SPARC ULTRA 1 non dédiée et qui supporte le système d'exploitation SUNSOLARIS 5.6.

Une grande partie du code des modules algorithmiques que nous avons utilisés proviennent d'une bibliothèque de programmes de l'équipe ROBOTVIS [ROBOTVIS WEB Page] de l'INRIA.

Au niveau des performances temps réel obtenues sur cette architecture matérielle, elles sont identiques à celles obtenues sans avoir utilisé ORCCAD pour implanter la même application [Murcia 97], à savoir autour de $10Hz$ pour l'action de suivi passif (suivi sans mouvement de caméra) et autour de $5Hz$ pour l'action de suivi actif (suivi avec mouvement de caméra).
Ces résultats relativement insuffisants pour assurer un suivi de cible lorsque la dynamique de la scène est forte s'expliquent par le fait que l'architecture matérielle envisagée est non dédiée.

Si on s'intéresse maintenant à l'utilisation de l'environnement ORCCAD pour mettre en œuvre l'application de vision, les gains sont indéniables en comparaison à la même application implantée directement. En effet, ORCCAD propose un "cadre" de spécification qui systématise la construction des actions et ensuite des applications. Cette construction permet la validation de la partie logique des spécifications. Ensuite, la définition sous forme graphique des différentes actions qui vont composer l'application facilite la compréhension et le maintien de l'application. Au sein de l'environnement, les différents modules, tâches et procédures sont réutilisables. Enfin, il permet de gérer de manière stricte et transparente les contraintes temporelles et génère automatiquement le code temps réel de l'application en fonction de l'architecture matérielle.

6.7 Conclusion

Nous avons présenté comment nous avons mis en œuvre une version simplifiée de l'application de la "mamy-sitter" sous forme de TÂCHE VISION en utilisant le système ORCCAD. A ce stade de la description, nous n'avons pas encore montré en quoi ce système répond ou non aux besoins des applications de vision temps réel. C'est que nous allons faire dans le chapitre 7.

Chapitre 7

Adéquation ORCCAD - Vision "Temps Réel"

Dans le chapitre 5, nous avons présenté l'architecture ORCCAD et la méthodologie de développement associée.

Le chapitre 6 a montré comment nous avons mis en œuvre une version simplifiée (sans processus de raisonnement) de l'application de vision temps réel la "mamy-sitter", décrite lors de la présentation du contexte de ce travail (chapitre 2).

ORCCAD a été conçu pour répondre très particulièrement aux besoins de spécification, et d'implantation des applications *robotiques*. Or, ces besoins ne sont pas exactement ceux des applications de vision. Aussi est-il important de nous pencher sur l'étude de l'adéquation entre la méthodologie proposée par ORCCAD et la mise en œuvre des applications de vision qui nous intéressent.

Dans ce chapitre, nous analysons en quoi ORCCAD convient aux applications de vision temps réel, quelles sont les limites du système et comment nous avons adapté cette architecture aux spécificités de nos applications.

Dans un premier temps, nous analysons le système ORCCAD afin d'en recenser les points forts et les points faibles dans le contexte d'utilisation particulier qui nous intéresse, à savoir les applications de vision de type video-surveillance. Enfin, nous présentons notre contribution concernant les solutions que nous proposons pour pallier au mieux ces insuffisances, sans toutefois dénaturer la philosophie de mise en œuvre des applications robotiques propre au système ORCCAD.

7.1 Analyse du système ORCCAD vis-à-vis des besoins en vision temps réel

Dans cette partie, nous allons étudier les différentes caractéristiques du système ORCCAD afin de dégager les points forts et les faiblesses du système vis-à-vis des besoins de la spécification et implantation des applications de vision temps réel.

Cette étude est réalisée en analysant successivement les thèmes envisagés lors de l'état de l'art, à savoir : les traitements avec une attention particulière sur l'exécutif temps réel, la

supervision, les processus de raisonnement et l'intégration.

7.1.1 Les traitements temps réel

7.1.1.1 Gestion des algorithmes

Points Forts.

L'approche modulaire et hiérarchique. ORCCAD propose une approche modulaire et hiérarchique pour la programmation d'application robotique. La hiérarchie repose sur les concepts de TÂCHE ROBOT (ou TR) et de PROCÉDURE ROBOT (ou PR) qui identifient respectivement le niveau *fonctionnel* et *contrôle* de l'architecture.
Les TÂCHES ROBOT correspondent à des boucles de traitements temps réel le plus généralement périodiques qui ont pour but de gérer les commandes d'un robot. Les PROCÉDURES ROBOT correspondent à l'ordonnancement logique des actions définies au niveau fonctionnel, c'est-à-dire, les TÂCHES ROBOT.

L'interface entre ces deux niveaux se fait par l'intermédiaire de signaux logiques.

Il y a donc une séparation stricte entre la partie des traitements algorithmiques, c'est-à-dire, qui effectuent des calculs, (au niveau TR) et de la partie qui gère le contrôle logique des traitements (au niveau PR principalement).
Au niveau TR, le contrôle a été conçu de façon à être générique à tout type d'action, et peut être facilement paramétré pour le spécialiser en fonction des besoins particuliers de l'action ou de l'application.

La gestion explicite du capteur La ressource physique encapsule les traitements qui sont dépendants de la ressource robotique à commander et gère les traitements non portables de la TR. Cette gestion est très intéressante car elle isole la communication avec la ressource qui fournit les informations "premières" à traiter par la TR. En particulier, on peut facilement interchanger une ressource physique commandant une caméra avec une ressource physique qui gère une séquence d'images.

L'analogie TÂCHE ROBOT-TÂCHE VISION. Le concept de TÂCHE ROBOT et de TÂCHE VISION, définie au chapitre 4.2.3, sont des concepts assez proches, d'où l'intérêt de se baser sur l'environnement ORCCAD pour mettre en œuvre les applications de vision temps réel. La TV et la TR correspondent toutes deux à des boucles de traitements périodiques, lesquels traitements doivent répondre à des contraintes temporelles particulières liées à la dynamique de la scène.

Faiblesses.

Le typage des ports de données. Au niveau des ports de données, nous avons vu au chapitre 5.2.1.1 que les ports des modules algorithmiques sont de type simple : on peut transmettre d'un module à un autre des scalaires, des vecteurs ou des matrices. ORCCAD

ne prévoit pas des ports de type structuré. Or, les *images* manipulées dans le domaine du traitement du signal et de la vision sont très souvent des structures complexes.

Dans l'application que nous avons développée sous ORCCAD présentée au paragraphe 6.1, on a transmis la donnée *image* sous la forme d'un entier, qui représente en fait un pointeur sur la structure *image*. Ici, le but est d'éviter des recopies inutiles de données lors de l'échange de données à travers les ports des modules. Car, sur des *images*, ces recopies peuvent être rédibitrices au niveau des contraintes temporelles à respecter et sont des sources d'incohérence si la recopie se passe mal. Dans le cas des actions robotiques, les données échangées sont de plus petites tailles et donc la recopie n'est pas pénalisante au niveau du temps d'exécution.

Or, ce stratagème sur un entier n'est pas toujours portable sur un autre type de machine et n'est donc pas satisfaisant.
En effet, d'un type de machine à une autre, le codage des entiers peut varier et la valeur du pointeur désigné par l'entier peut être complètement faussée et donc être inutilisable.

ORCCAD ne permet pas au programmeur de définir un *type "utilisateur"*, qui référence de manière plus générique un type plus complexe propre aux besoins du programmeur. Alors que le langage MAESTRO [Turro 99] gère un type utilisateur de la même façon qu'ESTEREL [Berry 92], dans le sens où l'utilisateur doit fournir, avec le type, les fonctions d'affectation, d'égalité nécessaires au langage pour qu'il sache comment manipuler les données de ce type.

L'absence de mémorisation. Au niveau de la composition en modules lors de la conception d'une TÂCHE ROBOT, l'environnement ORCCAD impose un arrangement de modules sans boucle, à savoir, il est impossible de relier la sortie d'un module avec l'entrée d'un autre en amont qui participe au calcul de cette sortie (et qui serait utilisée au prochain cycle de calcul). Ceci est dû à l'algorithme d'ordonnancement d'exécution des modules par le système, qui ne sait pas traiter les boucles.

Or, ceci entrave quelque peu la gestion de filtrage qui sont fréquents en traitement d'image et en vision. Lors d'une opération de filtrage, on utilise en entrée une donnée dont la valeur obtenue à l'exécution précédente. On peut voir cette donnée comme une sorte de mémoire d'un port de donnée.

Dans l'application que nous avons développée sous ORCCAD présentée au paragraphe 6.1, il a fallu rajouter le module de mémorisation **`Mem`** (se reporter à la figure 6.2). Ce module permet de conserver l'image de l'instant précédent $t-1$ afin que le module **`Difference`** puisse travailler sur les images aux instants t et $t-1$. Par conséquent, il serait intéressant de pouvoir gérer de manière implicite cette *mémoire* sur les ports de données. D'autant plus que les langages synchrones de type flot de donnée tels que LUSTRE [Caspi 87] ou SIGNAL [Guernic 91] formalisent cette mémoire directement dans le langage en tant qu'opérateur du langage : **`pre(X)`** pour LUSTRE, ou **`X$1`** pour SIGNAL (pour plus d'information sur ces langages, se reporter à l'annexe B).

La réutilisabilité des modules. Dans le contexte des TÂCHES ROBOT, la ressource physique regroupe l'ensemble des traitements informatiques qui dépendent du robot, contrairement aux modules algorithmiques qui peuvent être réutilisés pour composer d'autres lois de commande avec autres ressources physiques. En effet, en robotique, les structures de

données manipulées par les modules algorithmiques sont très simples et ceci facilite la réutilisation des modules.
Cependant, en vision, les traitements sont fortement dépendants de la structure de l'image acquise, de son format, mais aussi de la façon dont les images ont été acquises. En effet, il est courant que les caméras disposent de cartes d'acquisition qui comprennnent des traitements câblées, donc très rapides, qui permettent certaines transformations de base sur l'image acquise, par exemple, lissage, seuillage, différence, filtrage.
Par conséquent, le module de vision n'est plus réutilisable quelque soit la ressource physique employée. Cette caractéristique met l'accent sur l'intérêt de disposer des bibliothèques de modules qui travaillent sur des structures de données génériques ou pour les traitements bas niveau de bibliothèques spécifiques à une ressource physique.

L'organisation des bibliothèques de modules. Comme il existe une bibliothèque de modules spécifiques aux robots manipulateurs, il serait intéressant de créer des bibliothèques de modules spécifiques aux traitements d'image et de vision. Le problème se pose de l'organisation d'une telle bibliothèque.

Souvent, les équipes de recherche dans ce domaine ont déjà des bibliothèques de traitements en fonction de leur besoin, L'opération qui consiste à retranscrire ces bibliothèques dans la structuration des modules ORCCAD peut paraître fastidieuse. Cependant, le module algorithmique ne comprend que du code entièrement réutilisable (puisque le code dépendant du matériel se trouve dans le module ressource physique). Ainsi, il peut être facilement échangé entre équipes mettant en œuvre des applications.

Une idée est d'organiser les bibliothèques selon la décomposition proposées par IUE [Mundy 93], TARGETJR [TARGET WEB Page] ou encore la bibliothèque XVISION [Hager 96].
En effet, ces environnements proposent la définition des structures des données et d'algorithmes standards pour le traitement d'image et pour les fonctionnalités de suivi (d'objet, de visage) suivant une hiérarchie de classes d'objets particulière. Par exemple, ces environnements décrivent des classes basiques tels que Image, Point, d'autres classes plus complexes comme Edge, Line, Pattern avec les méthodes correspondantes à chacun de ces objets.

La gestion des paramètres. Dans le chapitre 3, nous avons mis en exergue le fait que les traitements algorithmiques en vision sont fortement dépendants de paramètres qui vont influer sur la qualité des résultats produits et sur leur quantité, contribuant ainsi à altérer le temps de calcul. La gestion de ces paramètres est donc particulièrement importante.

Nous avons vu au chapitre 5.2.1.1 que le système ORCCAD gère des paramètres au niveau des modules algorithmiques composant une TÂCHE ROBOT :

1. la valeur du paramètre est fixe, c'est-à-dire, statique.
2. on lui associe un nom. La valeur du paramètre n'est plus fixée une fois pour toute, mais va pouvoir évoluer dynamiquement et ceci à travers la TR. En effet, ce nom va être associé à un paramètre de la TR. En conséquence, les paramètres dynamiques des modules algorithmiques composant une TR sont aussi les paramètres de cette TR.

   Pour modifier dynamiquement la valeur de ces paramètres,

– soit la valeur du paramètre est recalculée par le module algorithmique paramétré suite à une exception de type 1.
– soit elle est transmise à la TR lors de son lancement dans la PROCÉDURE ROBOT (paramètre qui peut provenir d'une autre TR).
– soit elle est donnée par l'utilisateur de manière interactive à travers un processus extérieur à l'application générée par ORCCAD, par exemple, une interface graphique. ORCCAD propose au processus extérieur voulant communiquer un paramètre à l'exécutable une interface sous forme de fonction. Cette fonction est produite automatiquement lors de la génération du code de l'exécutable. Sa signature est définie comme :

  int orc*nomTR***Prm([***liste des paramètres de la TR***])**

  où *nomTR* désigne de nom de la TR paramétrée, et
  [*liste des paramètres de la TR*] représente la liste typée des paramètres de la TR.

De plus, ORCCAD propose un mécanisme de reparamétrisation de la TÂCHE ROBOT en local à travers le mécanisme des exceptions de type 1.

Cependant, ces différents mécanismes ne sont pas suffisament flexibles pour gérer de manière adéquate les paramètres et par consèquent les traitements paramétrés.

Tout d'abord, ORCCAD ne permet pas de spécifier un intervalle de définition associé à un paramètre. Or, pour certains traitements algorithmiques de vision, on connaît a priori l'intervalle associé à un paramètre (de manière empirique). Donc le plus souvent on est capable de fixer les valeurs de l'intervalle. Un système comme HORUS [Schneider 97] permet ce genre de définition. En effet, ce système offre à l'utilisateur une gestion interactive des paramètres en générant un panneau graphique pour chaque paramètre de l'application. Les paramètres sont représentés par un curseur dont la largeur correspond à l'intervalle de validité du paramètre.

Ensuite, le mécanisme d'adaptation des paramètres, qui est enclenché par le module algorithmique paramétré après réception de l'avis de paramétrisation provenant de l'automate, gagnerait à être différencier du reste du code de ce module.
Le système à base de connaissance OCAPI [Thonnat 94], dédié au système de pilotage de programmes pour le domaine du traitement d'images, distingue la partie calcul d'un opérateur (ou module), l'évaluation des résultats de l'opérateur, l'ajustement des paramètres associés à un opérateur en fonction de l'évaluation, et leur initialisation.

De plus, le flot de données transformé par les TRs est globalement invariant durant l'exécution d'une TÂCHE ROBOT. ORCCAD ne sait pas gérer d'alternative d'exécution au sein des modules de traitements des TRs. Ainsi, il n'existe pas de mécanisme, par exemple, à travers un paramètre, qui permettrait de sélectionner le module algorithmique qui conviendrait le mieux en cours d'exécution, pour une fonctionnalité donnée (par exemple, les opérateurs de lissage tels que Sobel, Canny-Deriche [Deriche 87b]), et en fonction d'un critère particulier (temps d'exécution, fiabilité, etc.).

La gestion de la ressource physique. La ressource physique est un des composants essentiels de la TÂCHE ROBOT. La définition de la TR en ORCCAD induit qu'à chaque période, la ressource physique doit être commandée. Pour répondre à cette spécificité, un module al-

gorithmique qui envoie à la caméra une commande vide a été ajouté dans la spécification de la TR de détection de mouvement. Mais ceci alourdit la spécification de la TR inutilement par rapport aux besoins au niveau de la caméra.

Il serait intéressant de pouvoir distinguer différents types de ressources physiques. Trois catégories peuvent être envisagées.

1. *une et une seule* : la ressource doit être contrôlée en permanence, et n'est pas partageable par plusieurs TÂCHES ROBOT qui s'exécutent en parallèle.
   Par exemple : un bras robotique.

2. *au plus une* : la ressource est non partageable par plusieurs TÂCHES ROBOT, mais ne nécessite pas un contrôle permanent.
   Par exemple : des ultra-sons, la caméra ARGÈS lorsqu'on la commande en position (pan, tilt, rail).

3. *quelconque* : il n'y a pas de restriction sur l'utilisation de la ressource. Elle peut être partagée et ne requiert pas de contrôle permanent.
   Par exemple : la caméra ARGÈS lorsqu'elle est utilisée pour l'acquisition d'images et moyennant quelques précautions d'usage.

Cette classification a une influence sur :

- le protocole d'enchaînement des tâches. En effet, dans le cas 1, comme c'est le cas dans la version actuelle de ORCCAD, lorsque l'on enchaîne deux TRs, le système a mis au point une phase de transition, transparente pour le programmeur qui permet de contrôler la ressource physique commandé par une des TR jusqu'à la fin de l'initialisation de la suivante, sans provoquer de discontinuité sur la commande du robot qui pourrait lui être fatal vis-à-vis de la dynamique du système contrôlé (pour plus d'informations sur ce système de transition, consulter [Turro 99]). La figure 7.1 illustre ce phénomène.
  Classiquement, lors de la mise en séquence de deux actions, la fin de l'action 1 est synchronisé avec le début de la phase d'initialisation de l'action suivant 2.
  En ORCCAD, cette synchronisation n'est plus aussi simple. Lorsque l'initialisation de l'action 2 est lancée, l'action 1 s'exécute encore le temps que l'action 2 ait finie son initialisation et que l'exécution réelle ne commence. Ainsi, le robot pendant la phase d'initialisation de l'action 2 est encore contrôlé par l'action 1.
  Pour les cas 2 et 3, cette phase transitoire n'est plus nécessaire.
- la détection et mise en œuvre du type de transition correspondant à la génération de code MAESTRO → ESTEREL. En fonction du type de la ressource physique, on ne va pas gérer les transitions de la même façon et par conséquent, le comportement logique associé est différent.
- les vérifications sur les programmes MAESTRO : le programme spécifié par l'utilisateur respecte-t-il bien les contraintes sur les ressources ?
  En fonction de la ressource envisagée (*quelconque, au plus une, une et une seule*), il faut que le partage et le contrôle soient assurés en conséquence. Et ces particularités doivent se retrouver au niveau de MAESTRO et de sa gestion de l'enchaînement des actions sur les ressources.

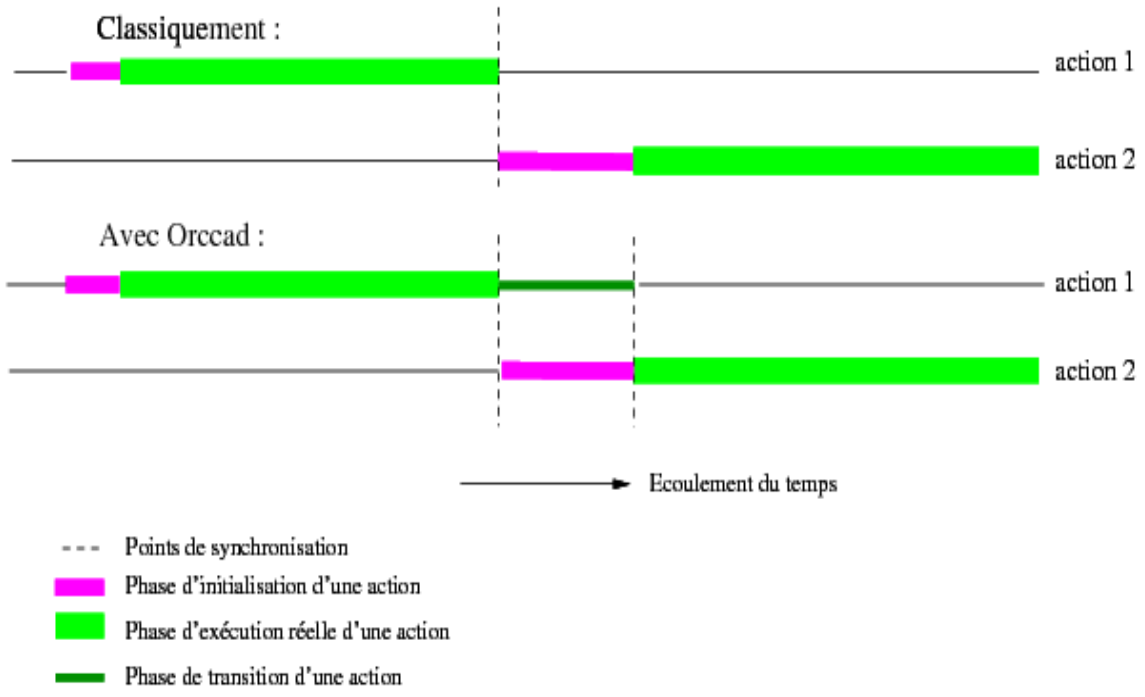


FIG. 7.1 – Synchronisation entre deux tâches successives

7.1.1.2 Vérification & Simulation

Points forts.

Les approches formelles. ORCCAD fournit une méthodologie dans laquelle les approches formelles sont omniprésentes.

En effet, l'entité de base, la TÂCHE ROBOT, implante une loi de commande basée sur la théorie de la commande et dont la stabilité a été étudiée au préalable. De plus, le comportement logique de l'application est exprimé suivant l'approche des systèmes réactifs synchrones. En particulier, la TR est caractérisée par un comportement logique générique qui peut être spécialisé par paramétrisation.

Cette utilisation de l'approche synchrone est très intéressante car elle permet des vérifications avec des outils mathématiques particuliers (par exemple, ATG [Roy 90] ou encore FC2SYMBMIN [Bouali 96]) qui vont permettre de valider ou invalider la partie contrôle logique de l'application.

La simulation. ORCCAD fournit aussi au programmeur les outils pour simuler l'application spécifiée.
Cette simulation peut être faite au niveau de la partie du contrôle logique de l'exécution de l'application. Pour ce faire, nous avons présenté au chapitre 5, paragraphe 5.2.3, l'outil XES [FC2TOOL WEB Page] qui est utilisé avec le programme ESTEREL représentant le comportement logique de l'application. La figure 6.12 donne un bon exemple de l'utilisation de cet outil pour la simulation de la partie contrôle logique. Cette simulation se fait par des clics

souris pour l'envoi de signaux au programme et on voit graphiquement quelle partie du programme est concernée ainsi que l'échange de signaux qui en résulte.
Alors que ATG permet de visualiser l'automate de l'application sur lequel on peut faire statiquement des vérifications en suivant les chemins entre états et transitions, XES permet de vérifier dynamiquement les propriétés du comportement logique de l'application, à travers le déroulement du programme ESTEREL correspondant.

ORCCAD propose aussi l'outil SIMPARC [Astraudo 92]. Cet outil permet de simuler plus complètement l'application générée par ORCCAD. SIMPARC propose de simuler conjointement la représentation des composants systèmes de la plateforme cible, de la ressource robotique utilisée, de la loi de commande, le tout intégrant le comportement logique de l'application.

Faiblesses.

La formalisation de la partie algorithmique. Alors que la partie comportementale est formalisée sous forme des systèmes synchrones, la partie algorithmique elle n'est pas soumise à une modélisation particulière, si ce n'est retranscrire sous forme mathématique l'équation relative à la loi de commande.

Cependant, la validation de la partie calcul est reportée à la simulation voire même à l'exécution. Cela modère l'intérêt des vérifications formelles mises à disposition par le système ORCCAD et qui ne concernent que la partie du contrôle logique de l'application. Il serait intéressant de disposer de techniques qui permettent automatiquement de vérifier certaines propriétés du code de calcul, comme les indices d'un tableau, la validité d'opérations sur des variables (telles que la division par 0).

Pour ce faire, on peut envisager d'inclure des techniques issues de l'analyse statique [Lacan 98]. L'analyse statique permet d'extraire automatiquement des propriétés sur des programmes écrits en langages généralistes. Ces propriétés sont une description formelle possible du comportement du programme (par exemple, la variable X est toujours impaire et première).
Ainsi, on est susceptible de vérifier automatiquement si, entre autres, certaines variables ont bien été initialisées, si les indices de tableaux ne sont pas erronés, ou encore si l'intervalle sur des nombres est bien conservé après un calcul (par exemple, sur les valeurs des ports d'entrée et de sortie).

L'analyse statique est un bon complément aux techniques de vérification comportementale de programmes, prévues dans la méthodologie ORCCAD.

Simulation et débogage de la partie algorithmique. Au niveau simulation, il manque des outils pour simuler efficacement les traitements d'image et de vision, car SIMPARC est mal adapté pour la modélisation de capteur et en particulier il est difficile de pouvoir modéliser une image acquise par une caméra simulée, le bruit qui s'ajoute à cette image.Actuellement, il ne modélise l'environnement que sous forme de polygônes. Il ne propose pas un rendu graphique proche d'une image vidéo réelle (qui prenne en compte des textures, des éclairages, des bruits capteurs, etc.).

D'autant plus que, le débogage de la partie du code inclus dans les modules n'est pas aisé. En effet, il faut se reporter directement au code généré par ORCCAD lorsque des erreurs de compilation surviennent. Il n'existe pas de relation entre la ligne de code généré et une ligne de code d'un module algorithmique.

7.1.1.3 Gestion des contraintes temporelles

Points forts.

La gestion transparente des contraintes temporelles. Un des grands points forts de ORCCAD est la gestion qui est faite au niveau des contraintes liées au temps réel.

En effet, le programmeur est entièrement déchargé de la partie programmation *temps réel* de l'application en fonction de la plateforme cible. ORCCAD permet automatiquement de générer un exécutable qui prend en compte ces spécificités. Le programmeur a juste à préciser le type de plateforme cible (VXWORKS, SOLARIS) et le code est généré en conséquence et en tenant compte des contraintes temporelles de chacun des modules algorithmiques composant l'application.

Cependant, pour l'instant, ORCCAD ne sait générer qu'un code monoprocesseur et monocadence.

Faiblesses.

La gestion des traitements de vision lents. Les applications générées sous ORCCAD mettent en œuvre des traitements périodiques, avec des contraintes temporelles fortes à respecter.

Or, en vision, on a à faire à des traitements lents, c'est-à-dire, qui requièrent des temps d'exécution qui ne sont pas compatibles avec des traitements temps réel : par exemple, les contours actifs, les algorithmes de reconstruction 3D utilisant de la stéréoscopie.

Ces traitements ne peuvent pas être inclus dans la chaîne de traitements périodiques. Cependant, il faut pouvoir les gérer si l'application globale en a besoin. Il faut pouvoir communiquer les informations que ce type de traitements nécessitent en dehors de la boucle de traitement. Or, ORCCAD ne dispose pas de canal de communication qui permettrait à des informations provenant de processus extérieurs de circuler vers la chaîne de traitements périodiques, et vice versa.

La gestion de la boucle de traitement trop stricte. Si la ressource physique ne reçoit pas de commande à la fin d'une période d'échantillonage, une exception de type 3 est émise et l'application est arrêtée.
Or, si la ressource est une caméra, le dépassement de temps d'une boucle de traitement n'est pas critique si il n'y a pas d'autres ressources plus critiques à gérer. Donc, la gestion du dépassement de temps (*overrun*) pourrait être rendue plus flexible pour les actions de perception.

Le temps d'exécution variable. En robotique, le temps de calcul d'une loi de commande est généralement indépendant des données. Ainsi, lorsque on a estimé le fréquence d'échantillonage convenant pour une loi de commande, elle reste valide quelque soit les calculs.
Or, en vision, nous avons vu au chapitre 3 3.1.1 que les calculs dépendent fortement des données en entrée. En effet, en fonction de l'image, de la scène, un module qui détecte des zones en mouvement fournit au module d'analyse de régions des listes de zones dont le nombre est variable. Par conséquent, le temps d'exécution dépend du nombre d'éléments qui seront à traiter.

L'idée pour gérer cette fluctuation du temps d'exécution c'est de limiter le nombre d'éléments à traiter. Et ceci peut être fait à travers des paramètres qui vont contraindre le nombre d'éléments à gérer et donc le temps d'exécution.

La parallélisation des traitements algorithmiques. Comme nous l'avons déjà dit au chapitre 3, il est important de pouvoir gérer la parallélisation des traitements algorithmiques.

Même si ORCCAD est prévu pour gérer des cadences multiples pour les traitements et de pouvoir distribuer le code des modules algorithmiques sur différents processeurs (voir le panneau de spécification des contraintes temporelles de la figure 6.5 du chapitre 6), la version actuelle de ORCCAD ne fonctionne qu'en mono-cadence et en mono-processeur.

En particulier, ORCCAD ne propose pas de mécanismes automatiques pour faciliter le parallélisme des traitements dans les modules algorithmiques comme c'est le cas pour le projet CAMERON [Najjar 98], ou encore le système SYNDEX [Lavarenne 91b].

Toutefois, des travaux ont été initiés autour de l'action incitative TOLÈRE
[TOLÈRE WEB Page] pour combiner ORCCAD et SYNDEX. Cette action a pour but de :

- proposer un environnement de programmation unique depuis la spécification d'un système jusqu'à son implantation temps réel optimisée.
- de prendre en compte les aspects liés à la tolérance aux fautes aussi bien au niveau logiciel que matériel, en exploitant la modélisation du contrôle de ces deux système qui peut s'exprimer sous un format commun, le format DC.

7.1.2 La supervision & l'IHM

Comme nous l'avons mentionné au chapitre 3, paragraphe 3.2, les applications de vision de type video-surveillance peuvent faire intervenir un processus de supervision sous l'aspect d'une interface graphique de type Interface Homme-Machine ou IHM.
Ce processus permet à un utilisateur d'être à la fois **informé** en temps réel, c'est-à-dire avoir accès à des données concernant l'exécution de l'application, mais aussi de pouvoir **influer** sur son déroulement en cours d'exécution.
Voyons comment ORCCAD permet de gérer ce type d'interface lors de la mise en œuvre d'une application.

Points Forts.

Deux niveaux d'interaction. Le système ORCCAD permet de discerner deux niveaux d'interaction entre un superviseur et l'application générée : un niveau données et un niveau contrôle. Ceci provient de la séparation stricte entre les données et le contrôle logique sur ces données proposée dans la méthodologie ORCCAD.

- Comment ORCCAD permet d'informer :
  - au niveau des données : lors de la génération du code de l'application, le programmeur d'application a la possibilité de désigner les ports de sortie des modules composant les différentes TRs de l'application, qu'il souhaite voir tracer vers un fichier, ou encore vers la sortie standard (se reporter à la figure 6.14 du chapitre 6).
  - au niveau du contrôle : de la même manière que pour les données, ORCCAD permet de tracer les événements définis comme signaux de sortie des modules composant l'application.
- Comment ORCCAD permet d'influer :
  - au niveau des données : comme nous l'avons vu dans le chapitre 5, paragraphe 5.2.1.1, une action robotique spécifiée sous forme de TÂCHE ROBOT peut être paramétrée. Ceci signifie qu'au lancement de la tâche, il faut fournir la valeur des paramètres attendus. Comme nous l'avons dit précédemment 7.1.1.1, ORCCAD permet de fournir cette valeur en cours d'exécution à travers une fonction particulière, qui est obtenue automatiquement au moment de la génération du code de l'exécutable ORCCAD. Nous rappelons que la signature de cette fonction est : **orc***nomTR***Prm([***liste des paramètres de la TR***])**
  - au niveau du contrôle : la spécification de la PROCÉDURE ROBOT peut faire appel à des événements extérieurs à l'application :
    - si on spécifie directement en langage ESTEREL, l'environnement ORCCAD propose le champs **Conditions** inclus dans le panneau de spécification de la PR (se reporter à la figure 6.8 du chapitre 6).
    - si on spécifie en langage MAESTRO, on dispose du mot clé **EXTERN :** (se reporter à la figure 6.9 du chapitre 6).

    Le système ORCCAD fournit une interface particulière pour communiquer les événements à l'automate global par le biais d'un appel à une fonction dont la signature est défini comme : **aa_I_***Signal***()** où *Signal* désigne le nom d'un événement à transmettre et qui a été spécifié soit au niveau PR soit au niveau TR.

Faiblesses.

L'interface avec un processus de supervision. La spécification des données et des événements à tracer se fait de manière statique avant la compilation du programme automatiquement généré, qui implante l'application.
Par conséquent, on ne peut pas dynamiquement échanger les informations (données ou signaux de sortie) que l'on veut tracer. De plus, outre les événements et les paramètres qui spécialisent les TRs, l'échange de données continu entre TR, ou provenant d'autres processus non ORCCAD n'est pas modélisé actuellement sur la version du système.

L'interface fournie par ORCCAD, c'est-à-dire, les fonctions qui permettent d'envoyer un événement à l'automate de l'application ou un paramètre à une TR en cours d'exécution

depuis l'extérieur de l'application sous-entend d'instancier l'objet implantant l'IHM directement dans le code de l'application. Par conséquent, cette technique lie l'application à l'objet IHM de manière statique. On ne peut pas lancer l'un sans avoir l'autre processus.
On perd donc en modularité, on limite la réutilisabilité de l'application avec un autre type d'interface graphique, on ne peut plus faire évoluer l'interface graphique indépendamment de l'application, c'est-à-dire, changer la structure de l'IHM qui permet la supervision, le type, etc.

De plus, le mécanisme d'interface proposé par le système via des fonctions pour les paramètres ou les événements n'est pas intuitif. Et le manque de documentation précise à ce sujet fait que le programmeur doit aller consulter le code généré par ORCCAD et trouver les fonctions particulières de manipulation de paramètres et d'événements construites automatiquement à partir des spécifications.

Ces différents points posent le problème de la gestion par le système ORCCAD d'une IHM.

Solutions envisagées On peut envisager différentes solutions, qui dépendent de la "portée" de l'IHM, selon qu'elle est gérée au niveau de la TÂCHE ROBOT ou au niveau de toute l'application.

- spécifier l'IHM comme une TÂCHE ROBOT en parallèle des autres TRs composant l'application nominale.
  La TR peut être composée de :
  – un module algorithmique.
    Dans **`init.c`**, il définit le type d'interface utilisateur, par exemple, X11, MOTIF, TCL/TK, ILOGVIEWS, JAVA, ou encore de l'ascii, le layout de l'interface, par exemple, les boutons, les menus. De plus, il faut définir la vie logique de l'interface avec des événements tels que une *postcondition*, qui indique la fin de l'interface, des *exceptions de type 2 ou 3* qui indiquent un arrêt urgent de l'interface, des *exceptions de type 1* qui indiquent un changement de mode de fonctionnement, par exemple, l'activation/désactivation de bouton, des affichages. Dans **`compute.c`**, on va définir les méthodes de visualisation pour les données images, les paramètres, les événements, ainsi que la manière de gérer des commandes "continues" provenant d'une souris ou d'un joystick.
  – une ressource physique fictive, étant donné que l'IHM est un objet qui n'est pas commandé.
  – un module automate qui va gérer les différents signaux définis précédemment.
  Comme les autres TRs, l'interface est lancée sous la forme d'un processus indépendant exécuté de manière périodique.
  Cependant, cette périodicité est assez accessoire pour une IHM, car une IHM est intrinsequement un processus interactif, non periodique.
  En revanche, on peut gérer le comportement spécifique de l'interface. En outre, intégrer le contrôle de l'interface graphique dans l'automate global de l'application permet d'effectuer des validations plus complètes car elles comprennent une partie du comportement logique de l'interface. Cependant, les événements tels que les clics souris correspondent à des signaux non classifiables dans la terminologie des TÂCHES RO-

BOT. Une idée pour pallier ce manque dans la typologie des événements ORCCAD est de rajouter une classe de signaux de type *synchronisation*, pour lesquels lorsque le signal est émis il n'a aucune influence sur la TR en cours mais sert à d'autres TRS.

- Traiter l'interface comme un module à part, c'est-à-dire, créer un nouveau type de module : le module interface utilisateur.
  On peut prévoir, dans la bibliothèque de module, de prédéfinir les modules correspondant à différents types d'interface. Le problème réside dans le fait que si l'application spécifie plusieurs TRS en parallèle utilisant une même interface, il faut faire apparaître le module d'interface dans la définition de chaque TR. Par conséquent, ceci entraîne une répétition de code, et de plus, cela pose le problème de la gestion de la cohérence du comportement de l'IHM.
- Gérer l'interface utilisateur comme une entité à part, non spécifiée par le système ORCCAD et que l'on va lier dynamiquement à travers un système de communication interprocessus (par exemple, des sockets). Il faut alors prévoir un protocole de communication générique entre le processus interface et l'application pour que les informations passent de l'un vers l'autre et réciproquement.
  Cette approche permet plus de souplesse au niveau de la gestion de l'IHM, et de l'application définie avec ORCCAD. Les deux processus sont indépendants. En revanche, on n'a pas facilement la possibilité de lier un comportement de l'IHM, par exemple, défini en ESTEREL avec celui de l'application ORCCAD.
- Sur ce thème de l'intégration des aspects IHM, il faut noter le travail préliminaire de [Kapellos 94]. Ce travail avait pour but de suivre le déroulement d'une application ORCCAD au niveau du flot de contrôle, en utilisant directement l'outil XES. Ce travail a été repris, dans le cadre d'un stage de DEA par E. Turci, encadré par D. Simon (équipe BIP [BIP WEB Page]), pour adresser le problème d'applications distribuées. Ce travail va plus loin qu'un couple (IHM - processus contrôlé), car il permet aussi de spécifier et d'implementer des processus temps réel s'exécutant sur des sites différents. La mise en oeuvre technique repose sur la répartition d'automates (avec l'outil OCREP [Caspi 94]), ainsi que sur la modification importante du noyau temps réel ORCCAD et en particulier de la partie qui gère la communication avec l'automate global généré pour l'application.

La solution que nous avons adoptée de mettre en œuvre correspond à gérer l'interface utilisateur comme une entité à part et par conséquent à adapter le système ORCCAD pour créer un canal de communication à l'exécution entre le processus généré par ORCCAD et l'IHM.

Nous reviendrons sur les détails de cette solution au paragraphe 7.2.3.

7.1.3 Les processus décisionnels

Points Faibles

Absence de niveau décisionnel. La version existante du système ORCCAD ne supporte pas de partie décisionnelle. Néanmoins, comme l'illustre la figure 5.1 du chapitre 5, il est possible d'interfacer une couche décision avec la couche contrôle, moyennant un travail de

formalisation similaire à celui des *vues abstraites* (voir le paragraphe 5.1.2) pour manipuler les entités de la couche contrôle.

Avec cette couche décisionnelle, il est envisageable de planifier des missions robotiques en manipulant des bibliothèques de PROCÉDURES ROBOT. En effet, l'obligation de prendre en compte les défaillances des actions élémentaires au niveau des procédures MAESTRO fait que ces dernières sont suffisamment robustes pour servir d'entités élémentaires pour un planificateur. En revanche, pour effectuer son raisonnement, un planificateur utilise une description de ses actions en terme d'objectif et de condition d'applicabilité. Il sera donc nécessaire d'ajouter cette information aux procédures MAESTRO si on désire les connecter à un planificateur.

Les travaux de [Kapellos 94] et [Gal 98] fournissent un début de réponse à cette problématique. Leur idée est de décorer les PROCÉDURES ROBOT avec des *préconditions* et des *postconditions*, indiquant leurs conditions d'applicabilité et d'élaborer un plan comme une succession de *précondition - postcondition* homogènes entre différentes PRs. Le plan se charge aussi de gérer les exceptions de type 2 et type 3 qui sont gérées au niveau de la spécification des PRs.

Cependant, outre ce travail sur ce niveau décisionnel, ORCCAD ne prévoit pas de mécanisme d'interfaçage entre l'application générée et des processus plus haut-niveau tel que des processus de raisonnement issus d'un système à base de connaissance. En d'autres mots, il manque une interface de communication entre un processus généré par ORCCAD et d'autres types de traitements qui ne peuvent pas être gérés par cet environnement (tels qu'un processus de raisonnement IA).

7.1.4 Intégration

Points forts :

Une approche globale de programmation d'application. Toutes les étapes pour mettre en œuvre une application sont clairement définies dans le système ORCCAD : spécification, vérification, simulation, exécution.

Pour chaque phase définie par ORCCAD, on dispose d'outils appropriés pour exploiter au mieux le système : une programmation graphique sous forme de schéma-blocs pour les TRs, le langage MAESTRO pour les PRs, les outils de vérifications utilisés implicitement (FC2SYMBMIN), explicitement (ATG), les outils de simulation (XES pour le contrôle logique, SIMPARC pour l'ensemble), un générateur de code automatique pour l'exécutable.

Faiblesses :

L'échange de données entre TÂCHES ROBOT. Deux TÂCHES ROBOT concurrentes définissent des flots de données disjoints. Un échange de données continu entre deux TRs n'est pas modélisé dans la version actuelle de l'environnment ORCCAD. Afin de "formaliser" d'éventuels échanges de données entre TRs s'exécutant en parallèle, il serait intéressant d'utiliser la notion de variable globale à une TR, ou plutôt de définir la notion de variable

au niveau de la PROCÉDURE ROBOT. Ce mécanisme permettrait de mettre en œuvre une communication asynchrone entre les deux tâches.

Ainsi, on localise rigoureusement la variable et ceci facilite la gestion de la synchronisation de type producteur-consommateur entre les TRs qui souhaitent communiquer entre elles. Une des TR produit et l'autre consomme la donnée, le tout étant synchronisé au niveau de la variable de la PR.

Avec la version actuelle du système ORCCAD, on peut gérer cette communication à l'intérieur du code des modules des TRs. Mais cela sous-entend de connaître la façon dont le système ORCCAD gère la liste des TRs actives en cours d'exécution et comment il accéde aux variables de ces tâches. Donc, il faut que le programmeur connaisse parfaitement tous les détails d'implantation du système lui-même, ce qui n'est pas du tout satisfaisant au niveau réutilisabilité.
Ceci pose le problème du formalisme à adopter au niveau PR et au niveau TR pour mettre en œuvre cette communication asynchrone de données. En particulier, le niveau PR ne manipule que des entités logiques, donc la variable globale devra être gérée comme telle.

De plus, ceci pose le problème de comment relier les deux tâches communicantes au niveau MAESTRO, et au niveau vérification, il faut assurer la cohérence de ces variables globales.

. La terminaison d'une TÂCHE ROBOT est consécutive à la réception d'un événement indiquant la bonne fin de la tâche (une *postcondition*) ou sa mauvaise fin (une *exception de type 2 ou de type 3*).
ORCCAD n'a pas d'autres types d'événements qui permettent la synchronisation entre différentes TRs à travers un signal, les *exceptions de type 1* étant traitées en local dans la tâche.

Ainsi, une TR ne peut transmettre de signal à une autre tâche et continuer son fonctionnement. Ce problème de synchronisation entre TÂCHES ROBOT qui s'exécutent en parallèle est particulièrement restrictive si on veut traiter et contrôler une action de type IHM au sein d'ORCCAD. En effet, une IHM envoie **tout au long** de son exécution des événements aux tâches intéressées et ce sans suspendre son fonctionnement.
Pour gérer ce problème, un idée est de rajouter un nouveau type à la classification des événements des TÂCHES ROBOT : ce nouveau type d'événement peut être émis par une TR à n'importe quel instant de son exécution et peut être reçu par toute TR en attente sur ce signal au niveau de la PROCÉDURE ROBOT.

En fait, cette synchronisation est possible directement à travers une *exception de type 1*. En effet, le principe de communication du langage ESTEREL repose sur la *diffusion instantanée* (ou broadcast) pour la transmission des signaux. Donc, lorsqu'un signal est émis, il est visible pour n'importe quelle partie de l'automate. Aussi, étant que l'automate généré comprend le comportement logique de toute l'application, une *exception de type 1* émise peut être reçu par une entité en attente. Cependant, une formalisation plus nette de ce mécanisme est nécessaire, en particulier pour les besoins de validation.

Le formalisme de spécification de l'application. Le langage MAESTRO est utilisé pour composer logiquement les applications robotiques sous forme de PROCÉDURES RO-

BOT. Ce langage est particulièrement dédié aux applications robotiques : ses opérateurs sont dédiés au contrôle, et les entités élémentaires manipulées sont des actions robotiques ou des événements.
De plus, il est basé sur une approche formelle ce qui permet de faire des vérifications aussi bien au niveau sémantique qu'au niveau de la logique du programme spécifié.

On peut envisager de spécifier les applications de vision avec d'autres formalismes. [Davis 97], [Pinhanez 99] ont utilisé le formalisme des *interval-scripts* pour spécifier une application complexe de vision. Un *interval script* est un langage de spécification d'interaction basé sur les concepts d'intervalle de temps et de relations temporelles. Il associe un intervalle temporel à chaque action du script. Pour chaque action, on définit les événements qui marquent le début et la fin de l'action. Et on peut aussi définir l'action avec les événements qui limitent son intervalle d'exécution.
On peut aussi citer SIGNALGTI [Rutten 95] qui est une sur-couche de SIGNAL, Ce système permet aussi de spécifier l'enchaînement d'actions de vision en utilisant une logique qui se base sur les intervalles de temps où le flot de données est valide. Alors que MAESTRO propose une programmation impérative, les applications spécifiées par les *interval scripts* ou SIGNALGTI se présentent sous la forme de règles ou de déclarations décrivant chacun des aspects de l'application, on parle alors de style de programmation déclaratif. Avec ce style de programmation, le programmeur n'a pas à se préoccuper de la suite de transformations élémentaires qui font passer des données d'une application à ses résultats. De plus, cette programmation se rapproche beaucoup des principes propres à l'Intelligence Artificielle.
Aussi, l'utilisation de ce type de formalisme haut niveau peut être intéressant pour rajouter un niveau proche du décisionnel, permettant ainsi de lier les actions avec des processus de raisonnement.

La communication avec d'autres types d'applications. ORCCAD est un système suffisamment ouvert pour que l'utilisateur puisse y intégrer ses propres modifications, en fonction de ces besoins.
Cependant, l'utilisation des résultats issus de ORCCAD sont difficilement exploitables par d'autres processus, tels que des processus de supervision ou encore de raisonnement. Nous avons vu au paragraphe 7.1.2, concernant la supervision, que la gestion d'un processus tel qu'une IHM n'est pas simple à mettre en œuvre au sein de l'architecture ORCCAD. En effet, ORCCAD ne fournit pas de point d'entrée pour faciliter la communication entre le processus généré et d'autres processus. Ceci est aussi vrai si l'on veut considérer plusieurs processus ORCCAD entre eux.

7.2 Adaptations réalisées

Nous allons présenter dans la suite de ce paragraphe les adaptations que nous avons mis en œuvre afin que le système ORCCAD réponde mieux aux spécificités des applications de vision qui nous intéressent.
Ces adaptations doivent être vues comme des extensions et pas comme des changements radicaux de l'architecture ORCCAD. Elles préservent les aspects liés au "contrôle-commande" propres à la philosophie *Orccadienne*, et visent à assouplir l'architecture pour prendre en

compte nos besoins particuliers de mise en œuvre au niveau de la vision, notamment en matière de reparamétrisation et de communication entre l'application générée par ORCCAD et des processus "autres" provenant d'autres types d'applications.

7.2.1 Au niveau de la spécification

Bibliothèques de modules. Une bibliothèque de modules dédiés aux traitements d'images "bas-niveau" a été rajoutée à la bibliothèque de modules algorithmiques déjà existante qui gérait des modules de commande pour robot manipulateur et robot mobile.

Cette bibliothèque concerne des traitements $image \rightarrow image$, tels que le lissage, la différence, ou encore le seuillage d'image.

Cette bibliothèque n'a pas été structurée suivant la décomposition définie par IUE ou TARGET. Et elle a le défaut de comprendre des modules dont le code inclut des appels proposés par la carte d'acquisition basée sur la bibliothèque XIL du système SOLARIS. Cette bibliothèque propose des fonctions performantes en temps d'exécution pour le traitement d'images. La XIL manipule l'image issue du capteur visuel suivant une structure propre et pour laquelle les fonctions de traitements d'images qu'elle propose sont optimisées. Ces fonctions effectuent très efficacement des opérations de base sur des images telles que multiplier une image par une constante, changer l'échelle d'une image, translater, filtrer, lisser par un filtre de Sobel ou encore seuiller une image.
Par conséquent, ces modules sont fortement dépendant de la carte d'acquisition, c'est-à-dire, du matériel. Or, selon la philosophie ORCCAD le code des modules doit se suffire à lui-même, c'est-à-dire, il doit pouvoir être utilisé sur une autre architecture matérielle.

Ce problème au niveau de la structuration de la bibliothèque pose un problème plus délicat. En ORCCAD, les parties dépendantes du matériel sont encapsulées dans le code propre à la ressource physique. Or, si on inclut ces traitements dans la ressource physique, ceci sous-entend de rendre la ressource physique paramétrable ce qui n'est pas prévu par le système ORCCAD. De plus, ceci rend la ressource physique beaucoup moins réutilisable. En effet, si au lieu d'un simple lissage de SOBEL, proposé directement par la bibliothèque XIL, on veut lisser avec un filtre de Canny-Deriche, il faut réécrire la ressource physique correspondante. Les modules utilisés en vision pour le bas niveau, c'est-à-dire, qui exploite directement les informations issues du capteur visuel, peuvent dépendre du capteur et de la carte d'acquisition. De plus, le format de l'image est différent en fonction du capteur.

Transition entre action. Le système de transition entre actions intrinsèque à l'architecture ORCCAD a été simplifié pour le cas des applications de vision envisagées.

La phase d'initialisation d'une action A_{i+1} et la phase de terminaison d'une action A_i ont été décorrélées. Ces deux phases se font en séquence dans le cas d'une caméra indépendante d'une autre ressource robotique qu'il faut commander à chaque période d'échantillonnage. D'autant plus que la caméra une fois initialisée, n'a plus besoin de l'être.

Gestion du paramétrage au niveau des TÂCHES ROBOT. Le mécanisme de reparamétrage des actions en cours d'exécution a été modifié pour répondre aux besoins des actions

de vision. Ce mécanisme permet l'estimation des résultats d'une action et l'adaptation de paramètres en conséquence pour assurer la robustesse des actions. Nous proposons de rajouter dans la structuration du module algorithmique un composant **`adapt.C`** qui implante le calcul de l'adaptation des paramètres du modules. Ce composant est appelé lorsque une reparamétrisation est nécessaire. Cette reparamétrisation est effectuée par un module particulier qui a la charge d'estimer les résultats issus de l'action. Il est lié à l'automate et lui transmet les événements de type 1 modifiés. En effet, ces événements contiennent une information supplémentaire qui est l'identification du module qui doit se reparamétrer. Cette identification provient du module d'estimation des résultats. Ce mécanisme évite à l'utilisateur de spécifier pour chaque exception de type 1 envoyée à l'automate la donnée associée émise par l'automate indiquant qu'un module doit être repamétré.

Ce système permet aussi de gérer plus facilement une demande de repamétrisation faite par un processus extérieur (processus de raisonnement). On paramètre le module d'estimation des résultats de façon à lier dynamiquement ce module et le processus extérieur de raisonnement, lors de l'exécution de l'action.

Les codes d'estimation et d'adaption sont à implanter par l'utilisateur. On pourrait penser à proposer à l'utilisateur un certain nombre de méthodes d'adaptation, ou à conserver les méthodes dans une bibliothèque.
Mais ceci pose le problème épineux des performances des algorithmes de vision et de leur comparaison pour mettre en place des tests d'évaluation de performance. [Förstner 96, Christensen 97] formulent différentes propositions en faveur de la caractérisation des performances des algorithmes de vision. En particulier, ils proposent d'utiliser des mesures statistiques (telles que la variance) pour caractériser de manière homogène la performance des algorithmes.

Cependant, reste à savoir comment gérer le choix des modules partageant la même fonctionnalité. Une idée est de créer un *méta-module* répondant à une fonctionnalité particulière représentée par un certain nombre de modules algorithmiques. On peut envisager d'instancier ce *meta-module* dynamiquement en fonction d'un identifiant particulier représentant un module spécifique. Cet identifiant de module peut être obtenu en sortie de la fonction **`adapt.C`**. Toutefois, ce *meta-module* pose tout de même le problème de la définition du module générique. En effet, chaque *meta-module* doit comporter un ensemble de ports de données, paramètres ou encore port événement sortie et entrée qui est fixe, quelque soient les modules qui répondent à la fonctionnalité.

7.2.2 Au niveau de l'exécutif temps réel

La caméra présentée au chapitre 6.6 est interfacée avec une station gérée par le système d'exploitation SUNSOLARIS. Or au départ, ORCCAD ne supportait qu'un seul type de plateforme cible, à savoir VXWORKS. Par conséquent, il est apparu nécessaire de porter la partie du système ORCCAD qui gère automatiquement les appels systèmes temps réel sur un nouveau système d'exploitation,

Afin de rendre ORCCAD le plus extensible possible, c'est-à-dire, gérer différentes plateformes cibles de manière uniforme, il a fallu modifier la structure interne du système ORCCAD.

Pour ce faire, nous avons isolé rigoureusement chaque appel lié au système d'exploitation dans la structure interne d'ORCCAD. Cet inventaire nous a permis de déduire une bibliothèque générique des appels système qui sont instanciés dans la partie ORCCAD, qui gère la génération de l'exécutable. La bibliothèque concerne la gestion de ressources telles que les *sémaphores*, les *queues de message*, les *processus légers* ou *thread*, la gestion de *timers*, la gestion des *signaux d'interruption*.
Cette bibliothèque doit être spécialisée pour chaque plateforme temps réel nouvelle sur laquelle une application générée par ORCCAD doit être exécutée. De plus, elle facilite l'installation de nouvelles plateformes cible au niveau ORCCAD.

Pour rendre possible la généricité de cette bibliothèque d'appels, nous avons construit la bibliothèque d'appels de manière à ce qu'elle réponde aux recommandations de la norme POSIX ou **P**ortable **O**perating **S**ystem **I**nterface [Gallmeister 95]. En effet, de nombreux systèmes d'exploitation, tels que SUNSOLARIS 5.6 ou encore le système temps réel VXWORKS, supportent cette norme.

La norme POSIX propose la spécification d'un ensemble de fonctions permettant de solliciter les services de base d'un système d'exploitation. L'objectif est de garantir le développement d'applications portables au niveau du code source, entre les systèmes d'exploitation conformes à la norme.

Cette norme ne constitue pas la définition d'un système d'exploitation. Elle fournit une liste de points d'accès aux services du système : pour chaque fonction, le comportement attendu dans les différentes circonstances susceptibles de se produire est complètement défini. Le but de cette norme est de masquer les spécificités des systèmes sous-jacents.

POSIX fournit des moyens, sous forme de bibliothèques, pour rendre portables des applications écrites pour un système d'exploitation donné, et de rendre l'écriture d'un programme plus simple.

Cette norme a été standardisée par ANSI et ISO sous le nom POSIX 1003.1X, où le X correspond au type de l'extension envisagée : POSIX.1003.1b correspond à l'extension temps réel et POSIX.1003.1c à l'extension thread.

Donc, cette bibliothèque d'appels facilite le portage du système ORCCAD vers d'autres systèmes d'exploitation temps rèel.
Pour l'instant, ORCCAD supporte les systèmes VXWORKS 5.2 et 5.3 ainsi que SUNSOLARIS 5.6.

7.2.3 Au niveau intégration et supervision

Un point important, qui est apparu de manière récurrente dans l'analyse, que nous avons présenté dans ce chapitre, c'est le besoin d'ouvrir le système ORCCAD vers l'extérieur. Plus précisément, il s'agit de faire communiquer un processus issu du système ORCCAD avec d'autres processus qui ne peuvent pas être gérés directement sous ce système, à savoir des interfaces graphiques, des processus de raisonnement issu d'un système à base de connaissance.
Nous présentons ici succintement les adaptations que nous proposons pour la communication avec la mise en place d'un *proxy*, ainsi qu'au niveau de la supervision de l'exécutif temps réel ORCCAD, avec la génération automatique d'interface graphique qui permet de piloter le processus ORCCAD. Ces adaptations vont être présentées en détail au chapitre 8.

Architecture de Communication. Au paragraphe 7.1, nous avons vu de manière récurrente apparaître le besoin de faire communiquer l'exécutable généré par ORCCAD avec d'autres traitements tels qu'une IHM ou un processus issu d'un système expert.

Pour pallier ce manque, nous avons ajouté un canal de communication au sein de l'architecture du système ORCCAD. Ce canal permet à l'exécutif temps réel généré par ORCCAD de communiquer des informations à d'autres processus application de vision temps réel avec des processus de supervision interactifs tels qu'une interface graphique ou des processus de vision lent qui n'ont pas de contraintes temporelles à valider.
Nous proposons une architecture de communication basée sur un entité que nous appelons *proxy* et qui va permettre à un processus ORCCAD d'échanger des informations avec d'autres processus.

La figure 7.2 illustre l'architecture de communication que nous proposons.

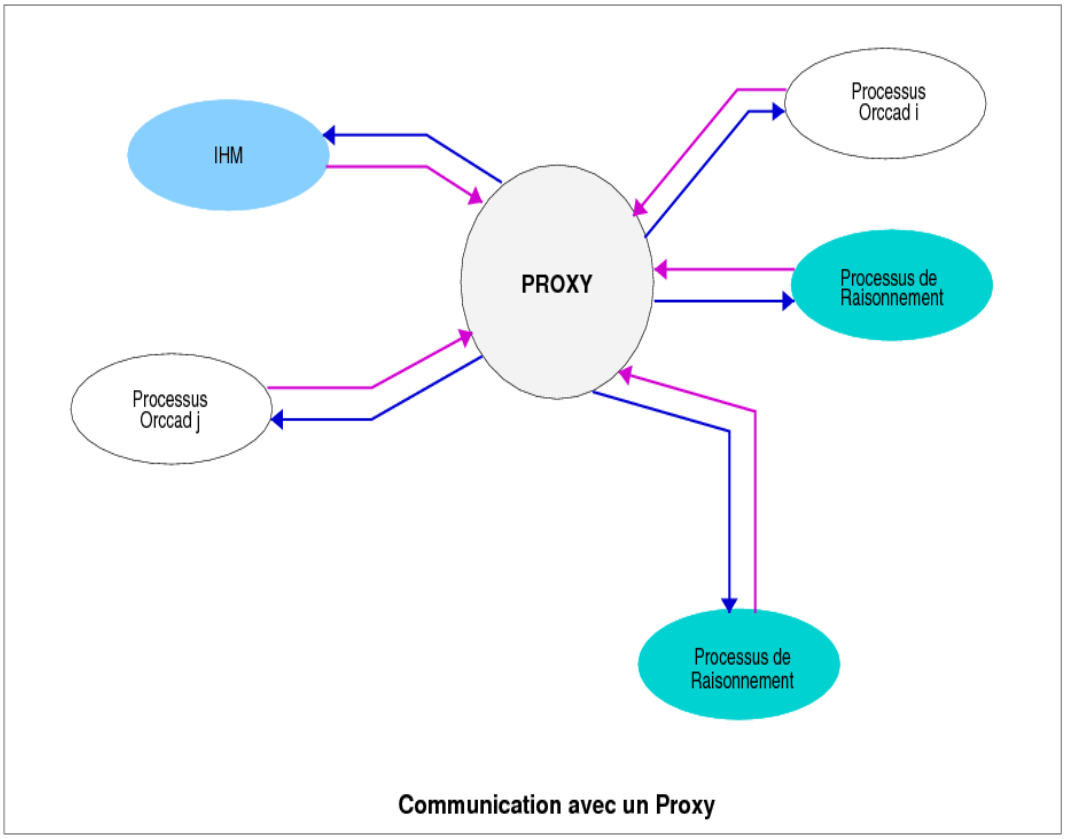


FIG. 7.2 – Architecture de communication proposée

Générateur d'interface homme machine pour la supervision de l'exécutif généré par ORCCAD. Au niveau de la supervision des exécutifs ORCCAD, nous proposons un générateur automatique d'interface graphique JAVA (utilisant la librairie graphique SWING). Ce générateur produit une IHM de supervision de la partie temps réel mise en œuvre avec ORCCAD. La supervision se traduit par des boutons de réglages pour les paramètres des TRs qui

compose l'exécutif ORCCAD, des boutons pour les événements ainsi qu'un panneau de visualisation de données des traitements temps réel sélectionnées par l'utilisateur. Un exemple d'interface générée automatiquement est donnée par la figure 7.3.
Cette interface permet aussi de palier au problème de la validation des intervalles des paramètres. Graphiquement, les valeurs possibles d'un paramètre sont limitées. Ces valeurs étant spécifiées par l'utilisateur au moment de la définition de la TÂCHE ROBOT et contenues dans le fichier de description qui permet de générer l'interface.

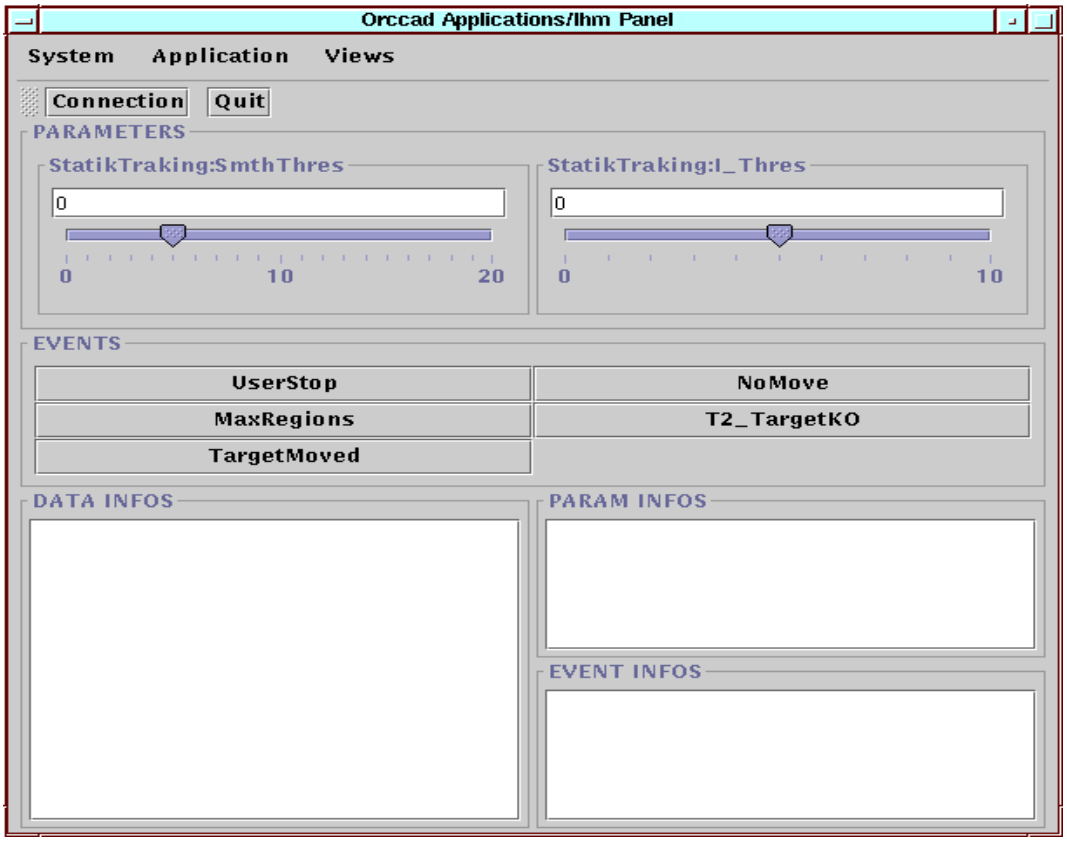


FIG. 7.3 – Exemple d'interface graphique généré automatiquement

7.3 Conclusion

Dans ce chapitre, nous avons analysé les points forts et les limites du système ORCCAD vis-à-vis des applications qui intéressent ce mémoire, à savoir les applications de type video-surveillance.

Cette analyse nous a conduit à mettre en œuvre les extensions suivantes dans le système :
– Au niveau des traitements temps réel : changement des transitions, proposition pour gérer le paramétrage au niveau module.

– Au niveau de la supervision : génération automatique d'une interface graphique (écrite en langage JAVA) qui gère les différents événements et paramètres de l'application, et permet de visualiser ces informations.
– Au niveau de la gestion de l'exécutif : création d'une bibliothèque d'appels générique. Elle facilite le support du système vers d'autres plateformes d'exécution en concentrant tous les appels "système d'exploitation"-dépendants.
– Au niveau intégration : création d'un canal de communication pour permettre de faire communiquer un processus généré par ORCCAD avec d'autres processus, tel que des processus IHM, de raisonnement ou d'autres processus ORCCAD.

La figure 7.4 synthétise les modifications apportées sur le système ORCCAD, les éléments en bleu localisent les entités de l'architecture à avoir subi des adaptations.

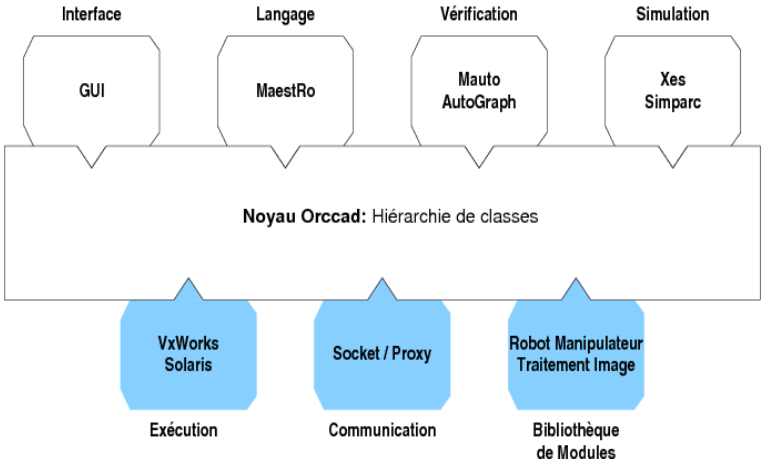


FIG. 7.4 – Nouvelle organisation de la boîte à outils ORCCAD, les contributions apparaissent en couleur.

Dans le chapitre 8, nous présentons en détail notre contribution concernant la gestion dans le système ORCCAD d'une communication entre le processus généré par cet environnement et d'autres processus .

Chapitre 8

L'architecture de communication

Nous avons vu, au chapitre 3, qu'une application de vision fait généralement intervenir différents processus, dont l'un traite des aspects temps réel.
Comme cet aspect temps réel doit être géré de manière rigoureuse, nous avons proposé d'utiliser le système ORCCAD.
De plus, nous avons mis en évidence, dans le chapitre 7, le besoin de faire communiquer la partie temps réel générée par le système ORCCAD avec d'autres processus.

Dans ce chapitre, nous proposons un mécanisme de communication ouvert, qui assure l'extensibilité de l'application.
Ce mécanisme se présente sous la forme d'une entité de communication intermédiaire, que nous appelons *proxy*. Ce proxy assure de façon transparente l'acheminement des messages entre les différents processus de l'application qu'il met en relation.

Dans un premier temps, nous revenons plus en détails sur les motivations qui nous ont conduit à ouvrir ORCCAD à d'autres processus. Ensuite, nous présentons l'architecture de communication que nous proposons, qui est basée sur la notion de proxy. Enfin, nous présentons de quelle façon nous avons mis en œuvre cette architecture de communication dans le système ORCCAD. De plus, nous proposons un mécanisme qui génère automatiquement une interface graphique de supervision à partir du processus temps réel créé par le système ORCCAD.

8.1 Motivations

Il peut être souhaitable, voire indispensable, de déplacer certains traitements sur une autre machine que la machine temps réel. Par exemple, les limitations de la machine temps réel peuvent ne pas permettre l'affichage de résultats intermédiaires ou d'images.
Par ailleurs, les ressources de la machine temps réel (telles que l'unité de calcul, la mémoire) peuvent ne pas être disponibles en quantité suffisante pour garantir l'exécution de tous les traitements en temps réel.
Heureusement, un certain nombre de traitements ne sont pas synchrones avec l'asservissement visuel et peuvent être délocalisés. Ces traitements asynchrones peuvent concerner les domaines suivants :

– **les programmes interactifs.** Les traitements en temps réel peuvent être paramétrés ou recevoir des événements de sources extérieures telles que les interfaces utilisateurs. Ces interfaces peuvent servir à la supervision de l'application, à la visualisation de résultats (en particulier, la visualisation d'images) et permettre, par exemple, de changer le mode de fonctionnement des traitements temps réel.
  Une IHM peut être très coûteuse en ressources, voire incompatible avec le matériel (par exemple, pas de ressources graphiques). Il est donc préférable de dissocier ces programmes interactifs de l'asservissement.
– **le débogage et les traces d'exécution.** Les traces sont un enregistrement de l'histoire d'une exécution, c'est-à-dire, la suite d'événements produits, les valeurs des paramètres, les valeurs intermédiaires de la boucle de traitements périodique, ou, plus généralement, toute information qui présente un intérêt au niveau de la vérification de l'exécution d'un programme. Elles sont utiles pour le débogage, en particulier lorsque les outils de débogage ne peuvent pas fonctionner correctement dans l'environnement temps réel.
  Pour pouvoir détecter toute singularité dans les traitements, il faut que la gestion des traces soit assurée de façon totalement décorellée de l'exécution temps réel. Comme pour les programmes interactifs, le débogage peut nécessiter des ressources qui pénaliseraient l'asservissement visuel, ou qui ne sont pas disponibles (par exemple, graphisme), voire un mode de fonctionnement incompatible avec un exécutif temps réel.
– **une architecture distribuée.** Il est possible que l'architecture du système ait été conçue de telle manière que certaines ressources physiques et les traitements en temps réel soient gérés sur des machines différentes, (les robots Romeo et Juliet [Khatib 96] sont un exemple représentatif de ce type d'architecture robotique). Dans ce cas, il faut fournir des mécanismes facilitant la gestion des ressources distantes.
  Il faut donc assurer une interface de communication à travers le réseau entre les différentes machines constituant la plateforme robotique de vision : par exemple, la machine qui permet de manipuler effectivement les capteurs et la machine sur laquelle l'asservissement visuel s'exécute.
– **les programmes non contraints temporellement.** Dans le cas d'applications de vision complexes, il est nécessaire de lier l'asservissement avec des traitements plus lents, qui n'ont pas de contraintes temporelles à respecter. C'est le cas notamment des traitements de vision "lents" et des traitements de type "raisonnement".
  Par traitements de vision "lents", nous entendons des traitements tels que l'algorithme de stéréoscopie pour la reconstruction en 3D d'une scène, ou l'identification de modèle. Les traitements de type "raisonnement" peuvent être issus d'un système expert dans lequel un moteur d'inférence effectue des raisonnements à partir des connaissances issues d'une base de connaissances.
  Or, ces programmes ne peuvent pas être pris en compte dans la suite de traitements périodiques, car les traitements de raisonnement peuvent entraîner le non respect des contraintes temporelles, liées à la dynamique de la scène, comme, par exemple, la perte de l'objet suivi. Par conséquent, il faut permettre à ces deux types de traitements de s'échanger des informations. De plus, ces échanges doivent être obtenus de façon transparente, c'est-à-dire, sans contrainte de programmation particulière, y compris lorsque les traitements s'exécutent sur des machines distantes.

Donc, une bonne architecture doit pouvoir gérer ces traitements annexes conjointement avec la boucle temps réel. Garder ces traitements au cœur de la boucle d'asservissement ne présente aucun intérêt. Au contraire, les placer à cet endroit risque de compromettre la modularité de l'architecture, la cadence d'exécution des boucles de traitements, et contraindre inutilement l'exécution de traitements non périodiques. Or, le système ORCCAD ne fournit pas de moyens pour lier ces deux types de traitements en dehors du cadre de la TÂCHE ROBOT.

8.2 Objectif

Pour pallier ce manque, notre objectif est de permettre aux processus générés par le système ORCCAD de communiquer de façon transparente avec les autres processus d'une application, par exemple, une IHM de contrôle, des traitements non temps réel, de type raisonnement, ou établir la communication entre deux processus ORCCAD. On veut réutiliser le typage des interactions propre au système ORCCAD. En effet, les données échangées entre les différents processus générés par ORCCAD sont les suivantes : événement, donnée, paramètre. Cette démarche permet de préserver un formalisme cohérent dans les communications vis-à-vis du système ORCCAD, et d'éviter aussi d'introduire des mécanismes de traductions de données qui pourraient être lourds et complexes à gérer. Par conséquent, tous les processus qui communiquent utilisent des messages typés suivant ces trois catégories.

8.3 Choix d'une architecture de communication

8.3.1 Solution Minimale

Le processus généré par ORCCAD établit (ou accepte) une connexion point à point avec chacun des processus avec lequel il est amené à communiquer. Ce mode de fonctionnement est suffisant quand le processus généré par ORCCAD n'a besoin de communiquer qu'avec un seul autre processus. La figure 8.1 schématise ce type de communication.

Mais, dès lors que l'on souhaite permettre à un processus généré par ORCCAD de communiquer avec plusieurs autres processus, ce type de communication est problématique, car il faut ouvrir autant de connexions que de processus à atteindre. La figure 8.2 illustre cette configuration. L'expéditeur doit, à chaque émission, déterminer à qui il destine le message. Or, nous cherchons à avoir une communication entre les processus qui soit transparente, c'est-à-dire sans désigner explicitement les destinataires du message.

Par conséquent, on doit faire en sorte que chaque processus, qui souhaite communiquer, ne gère qu'une communication point à point avec une autre entité distante. Mais il faut aussi que cette entité soit identifiée une fois pour toute, c'est-à-dire il ne faut pas contraindre la communication qu'avec une seule et même entité.

8.3.2 Solution plus flexible

On va utiliser un intermédiaire qui va être la seule interface connue du côté ORCCAD pour communiquer avec les processus non temps réel et inversement par les processus extérieurs

pour dialoguer avec le processus temps réel ORCCAD.
Cet intermédiaire, c'est un **proxy** ou encore **relais**. Ainsi, le processus ORCCAD ne connaît que le proxy et les processus extérieurs ne connaissent du processus ORCCAD que le proxy. Le processus ORCCAD ne gère donc plus directement la communication avec ces différents processus. Il transmet et reçoit des requêtes d'un intermédiaire qui gère la communication avec les processus extérieurs. De leur côté, les processus extérieurs ne voient l'application qu'à travers l'intermédiaire.
La figure 8.3 illustre la communication à travers un proxy.

8.3.3 Discussion

Ce principe de relais, intermédiaire ou de proxy est utilisé dans de nombreux domaines informatiques.

Dans le domaine de l'INTERNET, le *proxy server* [Luotonen 94] sert d'intermédiaire entre une machine utilisateur et l'INTERNET pour accélérer le chargement de pages WEB sur la machine utilisateur (système de cache)[1], et pour parer à toute intrusion venant de l'extérieur à travers INTERNET.

Dans le domaine de la conception orientée objet, le formalisme des "design patterns" [Gamma 95] préconise la notion de *patrons proxy* ou *mediator*. Ce *mediator* correspond à un objet qui sert d'interface d'appel à des fonctions réalisées par un groupe d'objets. Son rôle est d'organiser les interactions de ces fonctions pour répondre à des requêtes qui lui sont soumises.

Dans le domaine du pilotage de programmes, [Crubezy 99] a développé un *relais d'exécution* pour gérer l'exécution de programmes interactifs, distants, et les sous-parties d'un programme plus complexe. Le relais sert d'interface entre un *opérateur primitif*, ou tâche élémentaire, sélectionné par le pilote de programme et le programme externe qu'il représente. Le relais gère la communication et le contrôle à l'exécution, avec le programme externe associé à *l'opérateur*.

Dans le domaine de la robotique et plus précisément dans le cadre des systèmes "haptiques" (système à retour d'effort qui permet de toucher et manipuler des objets dans un environnement virtuel), [Ruspini 98] a développé un objet virtuel caractéristique, dénommé *proxy*, pour modéliser les interactions entre un utilisateur et son environnement virtuel. Dans ce contexte, le proxy représente un objet de taille finie et sans masse qui va se substituer au doigt de l'utilisateur ou à la sonde réelle dans l'environnement virtuel de l'haptique.

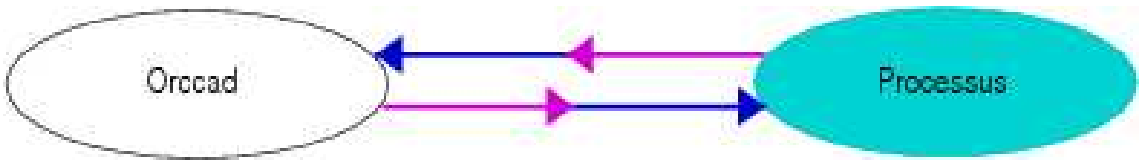


FIG. 8.1 – Principe de communication point à point.

[1] on parle alors de *proxy cache*

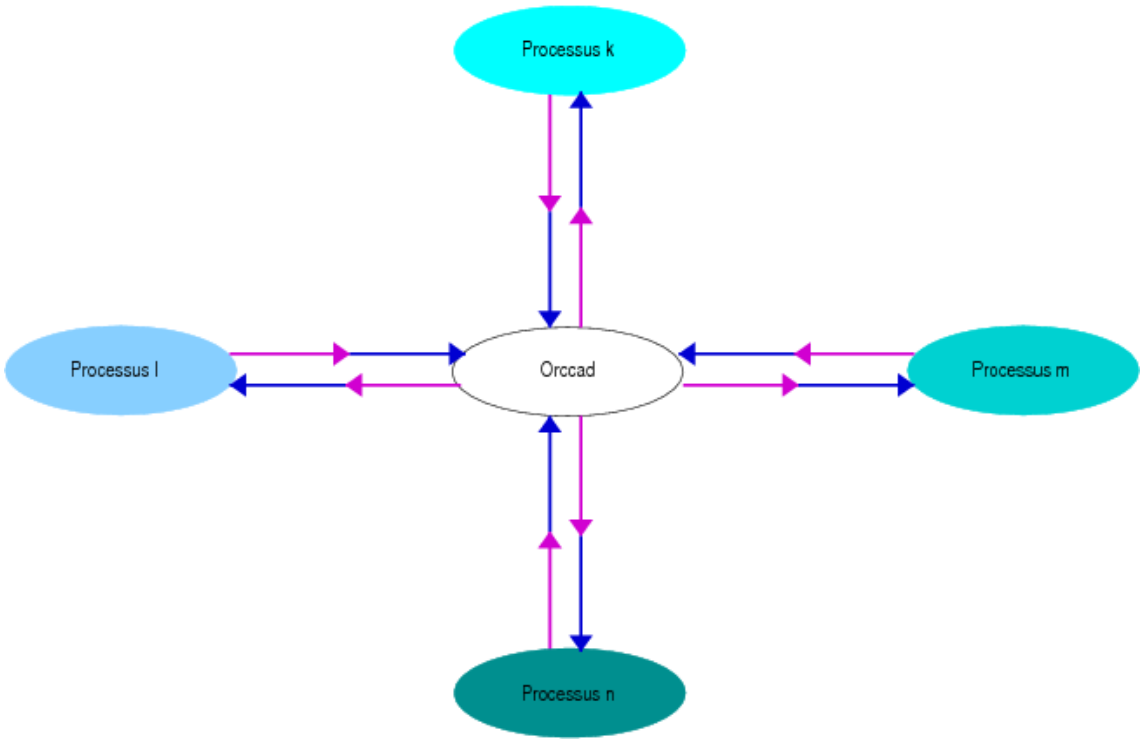


FIG. 8.2 – Schéma de communication basé sur de multiples communications point à point.

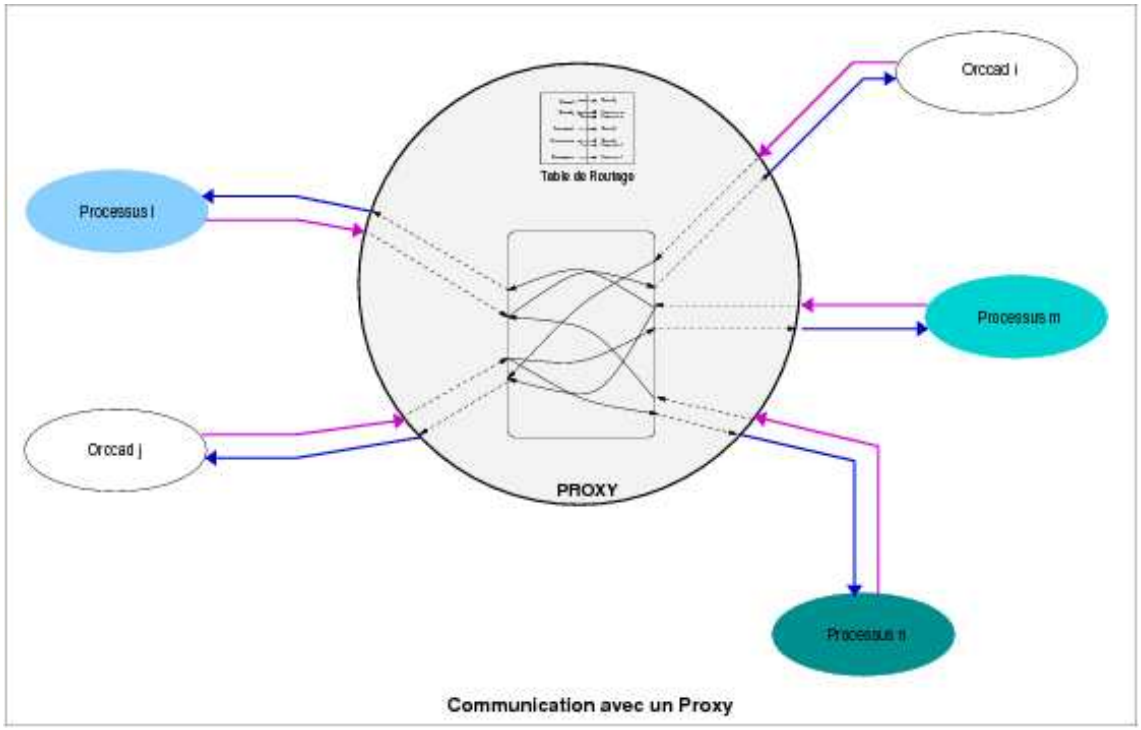


FIG. 8.3 – Architecture ouverte basée sur un proxy.

8.4 Mise en œuvre

8.4.1 Principe de fonctionnement

Le proxy est un processus indépendant qui joue le rôle d'un intermédiaire entre les différents processus qui veulent communiquer : d'un côté, les applications générées par ORCCAD, de l'autre les processus interactifs, distants ou encore de type "raisonnement".
Les processus se font connaître auprès du proxy en lui fournissant un identifiant, et en donnant la liste des informations qu'ils veulent recevoir des autres entités. Pour chacun des processus, le proxy établit une connexion distincte. Il gère une table particulière, que nous désignons par *table de routage*, qui lui permet de décider à qui adresser chacun des messages qui lui parviennent en fonction de leur nature.Cette table de routage établit la correspondance entre la *nature* des messages reçus et les connexions qui en sont destinataires.

En fait, nous souhaitons faire en sorte que chaque processus fasse comme si il communiquait en point à point avec un unique processus. Donc, il n'a pas besoin d'identifier explicitement le destinataire. Le processus envoie son message à un unique destinataire indépendamment du nombre réel de destinataires. Le proxy récupère ce message et se charge ensuite de le dupliquer autant de fois que la table de routage a désigné de destinataires. Pour les processus, l'action du proxy est transparente. La figure 8.3 illustre la prise en charge des communication par le proxy.

Le traitement des messages effectué par ce proxy est à mettre en parallèle avec la gestion des signaux en ESTEREL. En effet, le principe de communication d'ESTEREL repose sur la *diffusion instantanée* (ou *broadcast*) comme mécanisme de transmission des signaux. Par conséquent, dans un programme ESTEREL, les entités qui émettent ne savent pas qui va recevoir le signal et symétriquement, les entités en attente d'un signal ne savent pas quelle entité a émis cet événement. Toutefois, dans l'amélioration que nous avons apporté, le proxy transmet le message uniquement aux processus qui sont intéressés et non pas à la totalité des processus connectés. Le proxy implante donc des schémas de communication de type diffusion sélective (*multicast*) plutôt que diffusion globale (broadcast).

A titre d'exemple sur l'utilisation de ce type de mécanisme de diffusion, le système SOPHTALK [Clément 91, SOPHTALK WEB Page], utilisé dans l'environnement CENTAUR [Borras 88], donne une parfaite illustration de ce type de diffusion. Ce système, développé en LELISP, est un ensemble de fonctionnalités qui permet de décrire l'interaction d'objets selon un modèle de communication basé sur des événements. Il propose en particulier un mécanisme dénommé *stnode* de communication multicast. Les objets du système, dits *stnodes* émettent des messages lors d'événements significatifs. Ces messages circulent entre les stnodes et sont réceptionnés uniquement par les stnodes dont le type correspond aux messages. Ensuite, en fonction du message et du stnode, une action particulière est déclenchée.

8.4.2 Le protocole de communication

A présent, nous allons décrire le protocole de communication, c'est-à-dire l'ensemble de règles et de formats qui régissent les communications entre le proxy et les autres processus.

Dans ce protocole, nous distinguons deux types de message : les messages de transport et les message de contrôle.

Les messages de transport sont les messages de données utiles : ils sont pris en charge par le proxy qui les dirige vers leurs destinataires respectifs en fonction de règles de routage.
Les messages de contrôle s'adressent directement au proxy ; ils ne sont pas retransmis.

8.4.2.1 Messages de transport

Les messages reçus par le proxy sont composés de chaînes de caractères qui sont analysées afin de déterminer la nature du message.
Il existe trois types de messages de transport :

- le message donnée ;
- le message paramètre ;
- le message événement.

A chaque message, on associe un champ de date de composition du message, qui permet de déterminer sa durée de validité. Voici la syntaxe des messages de transport utilisée, décrite ci-dessus en notation "à la BNF" (Backus-Naur Form) [Backus 60] :

```
Message ::= Time:Type

Time ::= [h:mn:s]

Type ::= TypeDP|TypeE

TypeDP ::= DATA|PARAM:RTIdent:IdentDP:Val

TypeE ::= EVENT:ANY:IdentE

RTIdent ::= identificateur de la TÂCHE ROBOT | ANY

IdentDP ::= nom de variable de la donnée

IdentE ::= nom de l'événement

Val ::= valeur d'une donnée sous forme de chaîne de caractères
```

Quelques exemples :

[09:23:11]:EVENT:ANY:UserStop

[12:34:03]:DATA:ANY:NbRegions:5

[08:33:09]:PARAM:StatikTraking:SmtThres:3

8.4.2.2 Messages de contrôle

Ces messages servent à établir et à configurer les connexions à travers lesquelles les messages de transport sont acheminés.Ces messages sont transmis lors de la phase d'initialisation de la connexion. Ils permettent au processus de s'identifier auprès du proxy, et de configurer le routage des messages sur la connexion. Le proxy va affecter un identifiant particulier à chaque processus connecté.

Voici la syntaxe des messages de contrôle utilisée, décrite ci-dessus en notation "à la BNF". Selon que le proxy supporte ou non l'identification et la configuration dynamique des routes par les processus, il n'est pas nécessaire de supporter l'ensemble des messages présentés.

```
Message ::= Time:Type

Time ::= [h:mn:s]

Type ::= TypeId|TypeAuth|TypeRte

TypeId ::= IDENT:Pid

TypeAuth ::= AUTH:Key|Challenge:Val

TypeRte :: ROUTE:GetReq|DelReq|AddReq

GetReq ::= GET

DelReq ::= DEL:Rule

AddReq ::= ADD:Rule

Pid ::= identifiant du processus

Key ::= clé d'identification du processus

Challenge ::= chaîne codée pour l'identification du processus

Val ::= valeur d'une donnée sous forme de chaîne de caractères

Rule ::= règle de routage sous forme de chaîne de caractères
```

8.4.3 Le routage

La table de routage est l'entité qui permet au proxy de déterminer à quel processus adresser un message. Il s'agit, donc, d'une table de correspondance, entre l'ensemble des processus connectés au proxy et l'ensemble des types de message.

DATA :TRYYY :dataX	la donnée dataX de la Tâche Robot TRYYY
∗ :∗ :∗	toutes les données, paramètres, événements
∗ :TRXXX :∗	toutes les données, paramètres, événements de la TR TRXXX
PARAM :∗ :∗	tous les paramètres
EVENT :∗ :∗	tous les événements

TAB. 8.1 – Format des règles de routage

Nature du Message	Processus
DATA :TRYYY :dataX	Ident_Process$_i$
PARAM :TRXXX :paramY	Ident_Process$_i$
EVENT :∗ :∗	Ident_Process$_i$
∗ :∗ :∗	Ident_Process$_j$

TAB. 8.2 – Structure de la table de routage

Le proxy analyse le message afin de déterminer son type et le retransmet vers chacun de ses destinataires.
La table de routage peut être consultée suivant les deux canevas suivants :

i. pour chaque identifiant de processus, on associe la liste des types de message dont ils sont destinataires. On a l'association processus$_i$ → message$_n$
ii. pour chaque type de message, on associe l'identifiant du processus destinataire. La table gère l'association message$_i$ → processus$_n$

La solution (ii), illustrée par le tableau 8.2, est plus efficace à l'exécution mais un peu plus complexe à mettre en œuvre. Cependant, on peut très bien concilier ces deux canevas en générant automatiquement la table gérant le routage (ii) à partir d'une spécification fournissant une association processus$_i$ → message$_n$, c'est-à-dire suivant le routage (i).

La table de routage est construite à partir d'une liste de règles qui décrivent l'ensemble des messages qui doit être adressé à chacun des processus.

Ces règles de routage peuvent être construites à partir d'expressions régulières que nous présentons au tableau 8.1.

De nombreuses stratégies de construction et d'exploitation d'une telle table de routage sont envisageables. Selon que l'on désire privilégier la simplicité de mise en œuvre du proxy ou sa flexibilité d'utilisation, l'un ou l'autre des stratégies suivantes peuvent, par exemple, être choisies :

– **construction centralisée** : la table est construite à partir d'une liste préétablie des processus ou des types de processus supportés. Cette table peut être complètement figée (au début de l'exécution), ou recalculée à chaque nouvelle connexion.
– **construction collective** : dans ce cas, il n'existe pas nécessairement de liste préétablie (une liste par défaut peut exister). Ce sont les processus qui la configurent dynamiquement au fur et à mesure qu'ils se connectent au proxy. Ils utilisent pour cela les messages de contrôle de routage que nous avons vu au paragraphe 8.4.2.2.

8.5 Implantation

Nous présentons ici ce qui a été mis en œuvre au niveau ORCCAD.

8.5.1 Généralité

Pour mettre en œuvre la communication interprocessus entre un processus ORCCAD et les autres processus non temps réel, nous avons opté pour une architecture en "couche", c'est-à-dire on a adjoint à la partie applicative des processus une interface que nous appelons *talon* (stub en anglais) dédiée à la communication point à point. C'est cette interface qui gère de manière transparente les communications (envoi des messages, réception des messages). On dissocie ainsi la partie communication du reste des traitements du processus.

Le talon peut être utilisé de deux façons : (i) soit afin de mettre en relation deux processus applicatifs (voir la figure 8.4), (ii) soit afin de relier un processus applicatif à un nombre quelconque de processus en passant pas l'intermédiaire du proxy (voir la figure 8.5).

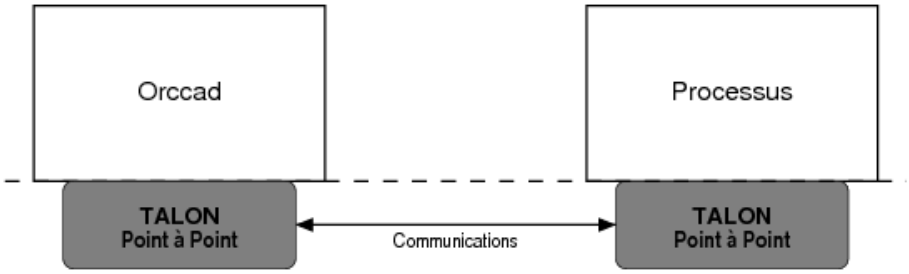


FIG. 8.4 – Architecture d'implantation en couches point à point

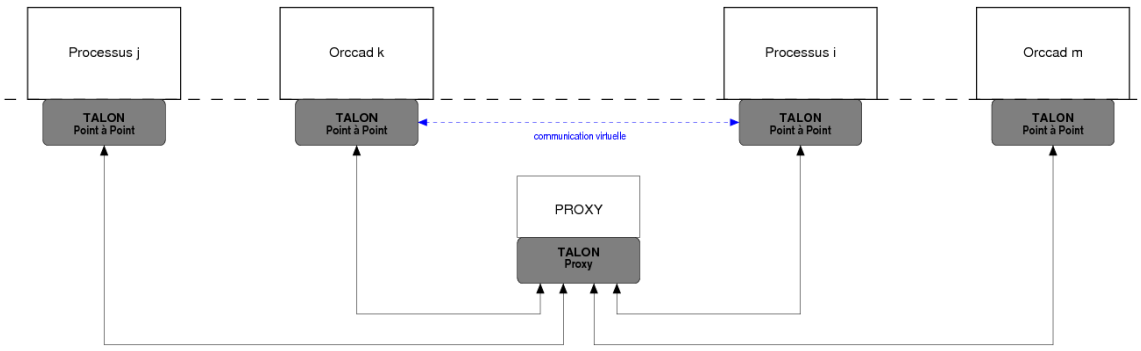


FIG. 8.5 – Architecture d'implantation en couches pour le proxy

Comme nous allons le voir dans le paragraphe suivant, cette notion de *talon* s'intègre très bien dans le système ORCCAD puisque ce dernier a été construit autour d'une hiérarchie de classes gérant, chacune, un point particulier du système (contrôle, exécutable temps réel, etc.).

8.5.2 Extensions du système ORCCAD

Pour gérer la communication dans ORCCAD, nous avons choisi de le faire à travers du mécanisme des *sockets*. En effet, une socket est une **API** [2] valable pour les échanges de données locaux ou distants à travers le réseau.

En effet, cette API est très répandue et permet l'échange de données aussi bien entre des processus qui s'exécutent sur une même machine qu'entre des processus qui s'exécutent sur des machines différentes, au travers d'un réseau. Dans ce dernier cas, de nombreux protocoles sont supportés dans ce dernier cas, à commencer par la suite de protocole TCP/IP, sur laquelle nous avons décidé de nous appuyer pour mettre en œuvre les communications entre processus.

Cette suite offre deux protocoles de niveau transport de données : TCP et UDP.
Le protocole TCP offre un certain nombres d'avantages : transmission fiable, séquencement des données garanti, contrôle de flux et de congestion. A l'inverse le protocole UDP ne propose qu'un minimum de garantie. Notamment, il n'assure pas au préalable que l'entité distante est bien présente. Par conséquent, l'utilisation de ce protocole est plus délicate que dans le cas TCP. C'est pourquoi nous avons choisi TCP pour la gestion de nos communications.

Au niveau de l'architecture logicielle que nous avons développée, nous avons choisi de :

- étendre la bibliothèque de fonctionnalités systèmes de ORCCAD pour supporter une socket de communication de type *serveur*.
- construire le proxy de telle sorte qu'il se connecte aux processus de type application ORCCAD à travers une socket *client* et qu'il gère une socket *serveur* pour les autres processus non ORCCAD.

Cette organisation s'explique par le fait qu'une application ORCCAD est le processus principal de traitement. Généralement, c'est lui qui va fournir les données que les autres processus vont exploiter. De plus, ceci permet de conserver côté ORCCAD la communication point à point entre une application ORCCAD et un autre processus, ce que ce soit le proxy ou un processus extérieur. Enfin, le fait de rendre une application ORCCAD *serveur* permet de la rendre indépendante d'une autre entité et de conserver le fonctionnement autonome de l'application.

Pour la gestion des connexions pour le proxy, elles sont traitées de manière statiques.

Côté utilisateur, la sélection des données de l'application ORCCAD à émettre vers l'extérieur se fait à travers le *menu de trace* qui liste pour chaque TÂCHE ROBOT tous ses ports de sortie données, paramètres et les événements. Un bouton a été rajouté au niveau du menu pour spécifier au système que les données sélectionnées vont transiter par le canal de communication. Ce panneau prévoit déjà la fréquence de l'émission des données, et le nombre de données à transmettre.
Le problème de cette technique, est que la sélection des données se fait avant la compilation du programme. Ceci rend impossible le changement des données à transmettre en dynamique en cours d'exécution.

Au niveau du système ORCCAD, l'ajout de ce canal de communication a sous-entendu l'extension de la librairie de fonctions temps réel pour comprendre les appels systèmes relatifs à l'établissement de la connexion, la déconnexion, l'envoi et la réception de messages à

[2] Application Programming Interface

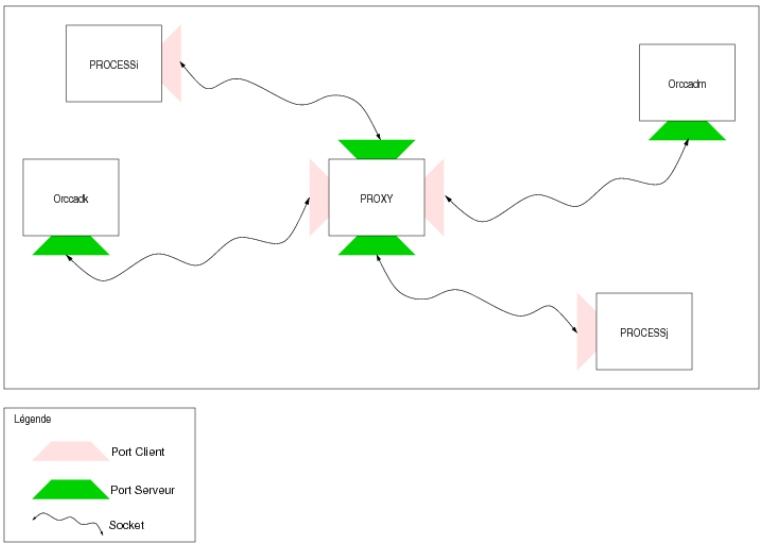


FIG. 8.6 – Implantation des communications sous forme de socket

travers une socket de communication. Pour ce faire, on a créé un objet `Communication`.
Au niveau de la connexion et déconnexion, l'objet `Communication` gère une socket *serveur* qui se met en attente d'une connexion *client* (provenant soit d'un processus non temps réel, ou du proxy). Pour la déconnexion, un message particulier est envoyé au client pour lui signifier la fin de la connexion et ensuite la connexion est fermée.
Pour la gestion des messages, l'objet `Communication` reçoit la chaîne de caractères correspondant au message à transmettre de l'objet gérant les traces, l'objet `Debug`. Pour les messages reçus de l'extérieur, l'objet `Communication` va analyser le message dans le but de déterminer le destinataire du message (quelle TÂCHE ROBOT ou l'automate). Pour ce faire, l'analyse va successivement récupérer le nom de la TÂCHE ROBOT, et le type de données (donnée, paramètre ou événement). Si c'est un paramètre ou une donnée, l'objet `Communication` récupère la liste des TÂCHES ROBOT actives et envoie le message à la TÂCHE ROBOT correspondante si elle est présente dans la liste. Si c'est un événement, l'objet communication utilise une méthode de l'objet `Canal` qui communique avec l'automate pour transmettre l'événement.
Tout comme les autres entités du système ORCCAD, tels que le canal de communication avec l'automate de contrôle, c'est un processus indépendant qui va gérer, à l'exécution, la communication. Elle se met en attente des messages en provenance de l'extérieur (proxy ou autre processus), et elle émet de l'information vers l'extérieur.
Il a fallu aussi modifier la gestion des traces d'exécution pour que les données sélectionnées et les événements puissent être envoyés à travers la socket de communication.

8.5.3 Génération automatique d'interfaces graphiques

Nous avons proposé une homogénéisation des communications entre processus autour d'un proxy et côté implantation avec l'utilisation du mécanisme de sockets. Nous avons

FIG. 8.7 – Panneau de sélection des données à transmettre

défini une classification précise des types de messages que les processus peuvent s'échanger, à savoir : donnée, paramètre, événement.

En fonction de ces considérations, il est très facile de générer automatiquement une interface graphique qui tire parti des communications à travers des sockets et de la classification des messages : sachant qu'un événement peut se représenter sous la forme d'un bouton, et un paramètre sous la forme d'un curseur. Ainsi l'utilisateur peut disposer gracieusement d'une IHM cohérente avec l'application générée par ORCCAD à travers laquelle il peut faire de la supervision.

Nous avons élaboré un programme JAVA, qui utilise la bibliothèque graphique SWING, permettant de générer automatiquement une interface graphique qui envoie des données de type paramètres, ou des événements particulier à un processus ORCCAD.

La génération se fait en fonction des spécifications de l'application ORCCAD.
On génère un curseur correspondant à chaque paramètre des TÂCHES ROBOT constituant l'application ORCCAD, et on génère un bouton pour chaque événement spécifié par l'utilisateur au niveau TÂCHE ROBOT ou au niveau PROCÉDURE ROBOT (événements externes spécifiables en MAESTRO). Cette interface contient aussi un panneau de visualisation pour les données, les paramètres et les événements produits ou émis par l'application (voir la représentation 8.4).

Les curseurs des paramètres sont définis par leur valeur maximale, valeur minimale, et valeur initiale. Cette manière d'opérer permet de contraindre les paramètres à un domaine de définition donné, qui peut dépendre de l'application ORCCAD envisagée.

La génération se fait en se basant sur un fichier de description de l'application qui fournit pour chaque TÂCHE ROBOT composant l'application : les paramètres et leurs caractéristiques, ainsi que les événements. Ce fichier est produit directement par le système ORCCAD

à partir des informations de spécification de l'application conçue en ORCCAD. La figure 8.3 présente la structure du fichier de description automatiquement produit par le système ORCCAD une fois l'application spécifiée.

```
Nom_de_TR
PARAMETERS
NOM_PARAM1 TYPE MIN_VAL MAX_VAL INIT_VAL
NOM_PARAM2 TYPE MIN_VAL MAX_VAL INIT_VAL
...
EVENTS
NOM_EVENT1
NOM_EVENT2
...
```

TAB. 8.3 – Structure du fichier de description pour la génération d'interface

8.6 Conclusion

Nous avons vu au chapitre 7 qu'ORCCAD ne dispose pas d'architecture permettant aux processus temps réel générés par ce système de communiquer avec d'autres processus.

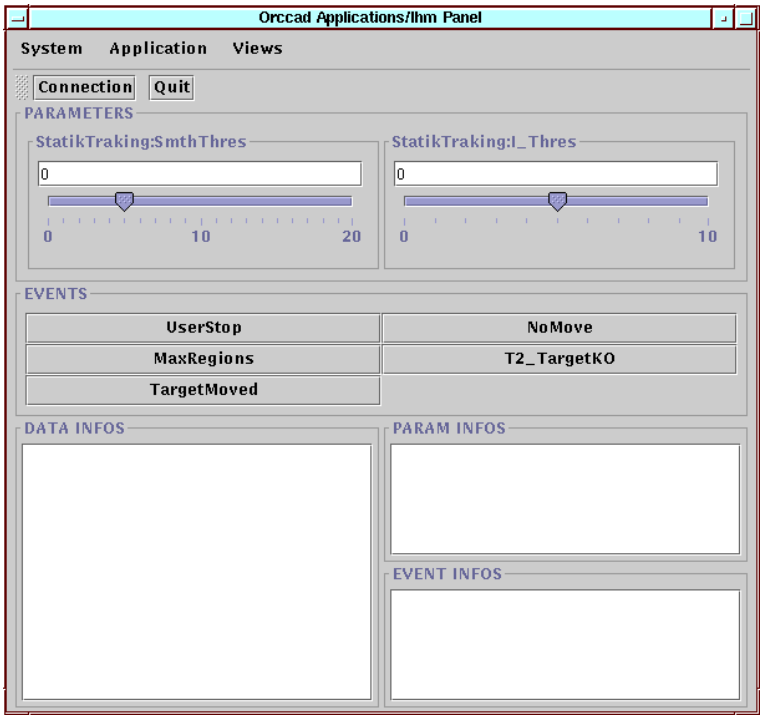
Ce chapitre présente notre solution pour pallier ce manque. Nous proposons de gérer la communication entre différents processus à travers un *proxy*. Ce proxy est une entité intermédiaire qui permet de gérer de façon transparente les communications entre les plusieurs processus.

Nous avons présenté l'architecture de communication que nous proposons basée sur le proxy. Nous avons décrit le protocole de communication des processus avec le proxy, la gestion des communications par le proxy et le fonctionnement du routage.

Nous avons montré comment nous avons implanté cette architecture dans le système ORCCAD au travers de l'API des sockets.

En plus de cette architecture de communication, nous avons décrit un mécanisme qui génère automatiquement une interface graphique de supervision correspondant à une application générée par ORCCAD. Cette interface contient de manière intrinsèque le mécanisme qui permet de communiquer avec le processus temps réel ORCCAD au travers du proxy.

StatikTraking
PARAMETERS
SmthThres INTEGER 0 20 5
I_Thres INTEGER 0 10 5
EVENTS
UserStop
NoMove
MaxRegions
T2_TargetKO
TargetMoved



TAB. 8.4 – **à gauche** : fichier de description d'une application qui va servir à la génération automatique d'interface graphique. **à droite** : interface graphique JAVA correspondante

Chapitre 9

Conclusions et Perspectives

9.1 Problématique

Dans le cadre de cette thèse, nous nous sommes intéressés aux applications de vision temps réel (de type monitoring vidéo), car ces applications sont représentatives des thèmes de recherche actuellement étudiés dans le domaine de la vision par ordinateur.
Ces traitements nécessitent la coordination d'une partie de traitement d'images en temps réel, d'une partie analyse de ces traitements qui prend des décisions par rapport à ce qui se passe dans la scène, et d'une partie de supervision qui contrôle le bon déroulement des traitements, et qui peut être pilotée de manière interactive, à l'aide d'une interface homme-machine. Par conséquent, ces applications requièrent une mise en œuvre qui assure leur robustesse et la sûreté de leur fonctionnement.
Pour ce faire, il est intéressant de disposer d'outils qui permettent formellement de valider la mise en œuvre d'un tel système et qui facilitent l'intégration de leurs différents aspects (temps réel, raisonnement, contrôle, supervision).

Nous avons choisi de traiter ce problème de conception et de mise en œuvre d'applications de vision temps suivant une approche *orientée tâche*. Nous proposons pour cela une formalisation de la partie des traitements temps réel, et une méthodologie de programmation tirant partie de cette formalisation.

9.2 Contributions

Nos contributions concernent une méthodologie de conception des applications de vision temps réel basée sur une hiérarchie de spécification, et l'extension de l'environnement ORCCAD/MAESTRO pour mettre en œuvre cette méthodologie.

9.2.1 Méthodologie de mise en œuvre d'applications de visions

Nous avons proposé une méthodologie de programmation de la partie traitement temps réel intervenant dans les applications de vision. Cette méthodologie permet de spécifier, et d'implanter les aspects temps réel des applications de vision. Cette méthodologie se base sur la description "formelle" d'une entité de base de programmation : la TÂCHE VISION.

L'analyse détaillée de différentes actions de vision utilisées pour mettre en œuvre des applications de suivi de cible en mouvement (voir le chapitre 4) nous a permis de caractériser cette TÂCHE VISION. Elle est décrite par un modèle hybride, dans lequel la boucle de traitements algorithmiques et la boucle de contrôle logique associée sont isolées.

Une TÂCHE VISION est donc formée des deux éléments suivants :

- une chaîne périodique des **transformations paramétrées** qui sont appliquées à l'information visuelle fournie par un **capteur**.
- un contrôle qui supervise au **niveau logique** l'exécution de la chaîne de transformations pour gérer le respect des **contraintes temporelles**.

A partir de ce modèle, nous proposons une méthodologie de programmation des applications de vision qui se fonde sur les deux niveaux d'abstraction suivants :

1. le niveau *tâche* (Tâche Vision). Ce niveau décrit un ensemble de tâches de vision et organise la structuration interne des tâches. Pour chaque TÂCHE VISION, une interface de manipulation générique, la *vue abstraite*, est construite à partir des informations logiques des tâches.
2. le niveau *application*. Les informations des vues abstraites, issues du niveau précédent, vont servir pour spécifier l'application sous la forme d'une arrangement logique de TÂCHES VISION. Les particularités de l'implantation sont ainsi masquées car les TÂCHES VISION ne sont manipulées qu'au travers de leurs vues abstraites.

9.2.2 Adaptation de l'environnement de programmation ORCCAD/MAESTRO [Team 98, Turro 99]

Pour mettre en œuvre cette méthodologie nous nous sommes appuyés sur un environnement déjà existant : ORCCAD/MAESTRO [Team 98, Turro 99].
L'environnement ORCCAD (voir le chapitre 5) est utilisé pour spécifier, valider et implanter des applications robotiques. Il comprend une méthodologie fondée sur deux niveaux : un niveau fonctionnel qui manipule les actions élémentaires de robotiques, appelées les TÂCHES ROBOT et un niveau application qui organise de manière logique ces actions élémentaires pour composer des PROCÉDURES ROBOT.
MAESTRO est un langage de programmation de mission, qui est dédié à la programmation d'applications robotiques. Il permet de spécifier les applications robotiques sous la forme des PROCÉDURES ROBOT.
Le système ORCCAD/MAESTRO propose un environnement de programmation graphique complet, permettant facilement de spécifier les différentes TÂCHES ROBOT, qui composent une application, de valider les applications au niveau logique et enfin de générer le code exécutable.

L'utilisation d'ORCCAD pour mettre en œuvre notre méthodologie se justifie car les formalismes des TÂCHE ROBOT et TÂCHE VISION sont relativement proches. Ils s'articulent tous deux autour d'une boucle temps réel périodique. Ces tâches distinguent rigoureusement la partie de calcul qui transforme un flot de données de manière continue, la partie flot de contrôle, et la ressource commandée. De plus, ORCCAD permet de valider formellement la partie relative au contrôle logique de l'application, car ce contrôle est exprimé sous la forme d'un programme ESTEREL.

Cependant, la différence la plus marquante entre TÂCHE VISION et TÂCHE ROBOT se situe au niveau des traitements considérés et de la gestion des paramètres, qui est un point spécifique de la TÂCHE VISION.

9.2.3 Faiblesses d'ORCCAD et Solutions apportées

Nous avons utilisé le système ORCCAD pour mettre en œuvre une première version de notre application test (la "mamy-sitter"), qui s'appuie sur le formalisme de TÂCHES VISION (voir le chapitre 6).
Cette application a nécessité une partie implantation des actions de vision pour la détection de régions, le suivi de cible, ainsi que la calibration de la caméra, qui sont basées sur des méthodes développées en collaboration avec l'équipe ROBOTVIS (INRIA Sophia).

L'environnement de programmation ORCCAD, qui est construit pour répondre aux spécificités des applications robotiques, a du être adapté pour gérer de façon adéquate les spécificités inhérentes aux applications de vision (voir le chapitre 7). Les adaptations réalisées sont les suivantes :

- Le mécanisme interne de transitions entre actions de l'architecture ORCCAD, conçu pour répondre au mieux aux besoins des applications robotiques, a pu être allégé, car les applications de vision sont moins exigeantes en matière du commande du capteur.
- Le mécanisme de reparamétrage des actions en cours d'exécution a été enrichi pour répondre aux besoins des actions de vision. Ce mécanisme permet d'adapter dynamiquement les paramètres d'une tâche à partir de l'estimation des résultats de ses modules algorithmiques et d'assurer la robustesse des actions.
- Une bibliothèque générique d'appels systèmes liés à la gestion du temps réel a été conçue pour isoler rigoureusement cette partie et faciliter le portage de ORCCAD vers d'autres systèmes d'exploitation temps réel. La bibliothèque gère les appels pour les systèmes VXWORKS (version 5.2, 5.3) et SOLARIS 5.6. De plus cette bibliothèque d'appels a été construite en suivant les recommandations de la norme POSIX.
- Une bibliothèque de modules dédiés aux traitements de vision "bas-niveau" a été développée. Elle enrichit la bibliothèque de modules algorithmiques déjà existante qui gère des modules de commande pour robot manipulateur et robot mobile.
- Une architecture de communication ouverte, qui assure l'extensibilité de l'application (voir le chapitre 8) a été développée. Elle se présente sous la forme d'une entité de communication intermédiaire, que nous appelons *proxy*. Ce proxy assure de façon transparente l'acheminement des messages entre les différents processus de l'application. Il permet, en particulier, de relier de façon transparente les boucles temps réel des traitements de vision avec des processus de supervision interactifs tels qu'une interface graphique ou des processus de vision lents qui n'ont pas de contraintes temporelles à valider et n'ont par conséquent pas besoin d'être implantés avec ORCCAD.
- Un générateur automatique d'interface graphique JAVA (librairie SWING) a été élaboré (voir le chapitre 8) pour fournir à l'utilisateur une interface homme-machine de supervision de l'application de vision générée à l'aide d'ORCCAD.

9.3 Conclusions et Perspectives

La méthodologie et l'environnement de programmation associé que nous proposons ont été validés expérimentalement sur une application simple de détection et de suivi de cibles en mouvement (voir le chapitre 6). Cette expérience a permis de mesurer l'intérêt de cette méthodologie pour le développement d'applications. Tout d'abord, ORCCAD propose un "cadre" de spécification qui oriente la construction des actions, pour permettre une construction correcte des applications. Cette construction permet la validation de la partie logique des spécifications. Ensuite, la définition sous forme graphique des différentes actions qui vont composer l'application facilite la compréhension et le maintien de l'application. Au sein de l'environnement, les différents modules, tâches et procédures sont réutilisables. Enfin, il permet de gérer de manière stricte et transparente les contraintes temporelles et génère automatiquement le code temps réel de l'application, pour différentes architectures matérielles.

Cependant, cette application ne couvre que partiellement les aspects participant à une application de vision, notamment au niveau des traitements d'analyse de scène, qui peut faire appel à des raisonnements nécessitant par exemple l'interrogation d'une base de connaissance.

Nous envisageons à l'avenir de traiter des applications plus conséquentes, comme par exemple, des applications de type "bureau intelligent", comme proposé par le système SMARTOFFICE [Gal 99]. Ce type d'application consiste en une pièce où on répartit plusieurs caméras qui vont observer les occupants pour les suivre, reconnaître et interpréter leurs activités telles que *entrer, sortir, s'asseoir, se lever* ainsi que des actions spécifiques faites sur un tableau noir.

Outre la validation expérimentale sur des applications plus complexes, il reste aussi des points à traiter au niveau de l'adéquation d'ORCCAD à la vision temps réel :

- **Au niveau des modules algorithmiques** composant la TÂCHE VISION, il serait intéressant d'étudier comment automatiser la parallélisation des traitements algorithmiques.
- **Au niveau de la validation**, l'approche permet de valider formellement la partie contrôle logique, grâce à l'utilisation du formalisme des systèmes réactifs et plus précisément du langage ESTEREL. Il serait intéressant d'étudier ce qui peut être fait au niveau algorithmique, et au niveau de la vérification du respect des contraintes temporelles. L'utilisation de techniques issues de l'analyse statique permettrait de vérifier automatiquement certaines propriétés sur les variables de programmes (dépassement d'index, division par zéro, etc). Des travaux sont engagés actuellement dans ce sens.
- **Au niveau de la gestion des paramètres.** Un autre point extrêmement important concerne l'évaluation des résultats issus des modules d'une tâche et l'adaptation des paramètres en conséquence. Nous avons vu que les paramètres et leur gestion dynamique étaient une caractéristique remarquable de la TÂCHE VISION.

  Jusqu'à présent, la communauté de vision s'est attachée à développer différents algorithmes de vision pour effectuer les traitements requis de façon la plus efficace possible. Cependant, chacun des algorithmes développés a des performances propres qui dépendent fortement de son contexte d'application. Il n'existe donc pas de "meilleur algorithme" applicable dans toutes les configurations. Il apparaît donc nécessaire de

systématiser l'étude de l'évaluation des algorithmiques [Förstner 96, Christensen 97], pour proposer des classifications d'algorithmes de fonctionnalité identique, et déterminer des procédures de choix qui permettraient de déterminer automatiquement, en fonction d'un "contexte" (à définir), quel algorithme appliquer.
L'accent doit aussi être mis sur l'étude de "lois de commande" sur les paramètres influant sur les traitements pour permettre l'évaluation des performances des différents algorithmes et pouvoir adapter les traitements en fonction des besoins des applications.

Annexe A

Comportements oculomoteurs : les concepts de base

A.1 Un modèle simple de caméra

Le système de vision active utilisé se base sur le système proposé par [Viéville 95]. Le modèle de la caméra est très simple puisque seuls trois paramètres (position du centre optique (u_0, v_0) et la longueur de la focale f) sont suffisant pour décrire la relation entre une point en 3D de coordonnées $M = (X, Y, Z)$ et sa projection 2D sur le plan rétinien $m = (u, v)$, donné en pixel (voir la figure A.2). La relation entre ces deux points est donné par :

$$u = f\,X/Z + u_0, \quad v = f\,Y/Z + v_0 \tag{A.1}$$

De plus, [Viéville 95, Willson 94] ont montré empiriquement qu'à ce niveau de précision (a) le centre optique (u_0, v_0) est presque fixe et varie aléatoirement lors d'un changement des paramètres de la caméra (focus ou zoom), et donc seule la valeur moyenne de ce point est statistiquement significative tandis que (b) la longueur de la focale f s'exprime comme une fonction linéaire du focus et du zoom.

L'approximation faite en (a) est aussi valide car on considère principalement des déplacements rétiniens tels que $\delta_t = (u_{t+\delta_t} - u_t, v_{t+\delta_t} - v_t)$, qui ne sont pas fonction de (u_0, v_0), on a simplement besoin d'une valeur estimée de ce point.

Ce modèle de calibration a été validé par [Viéville 95] and a été vérifié expérimentalement avec notre système, présenté à la figure A.1.

A.2 Une mosaïque comme représentation de données visuelles

Pour conserver l'information relative à une scène observée, on définit une représentation de l'image notée mosaïque, définie comme suit :

– on considère une représentation de l'environnement comme *cyclopéen et statique* , i.e. le système visuel est censé être positionné en un point précis de l'espace ;

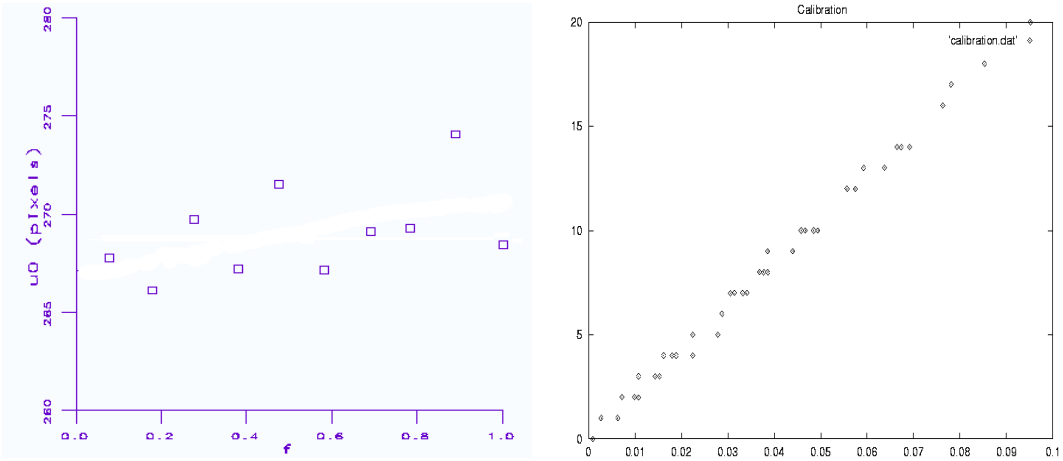


FIG. A.1 – Calibration du pan en pixel/radian : la variation du centre optique est inférieure à 10 pixels (à gauche) tandis que la linéarité du modèle par rapport à la focale est bien vérifiée (à droite).

– chaque "cellule" est censée correspondre à une orientation donnée de la direction du champs de vision de la caméra, définie dans le référentiel cyclopéen et repérée par l'angle horizontal H et vertical V, comme décrit à la figure A.2.

Etant donné que la surface de la mosaïque est régulièrement échantillonnée par les coordonnées angulaires de la direction du champs de vision de la caméra, sa structure différentielle est une surface sphérique.

Pour une résolution angulaire donnée, typiquement de $0.2\ deg$ et un champs de vue (avec rotations de la caméra) inférieur à $[\pm 100\ deg \times \pm 30\ deg]$, la taille de la mosaïque est de 1000×250 cellules, ce qui est parfaitement gérable.

La relation entre un pixel de l'image et une cellule de la mosaïque se calcule facilement, en considérant le modèle de caméra caractérisé par l'équation (A.1). En notant P, l'angle de rotation en pan de la tourelle de la caméra autour de l'axe vertical Y, et T, l'angle de rotation en tilt autour de l'axe vertical fronto-parallèle [1], on obtient :

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \equiv \begin{pmatrix} f(z) & 0 & u_0 \\ 0 & f(z) & v_0 \\ 0 & 0 & 1 \end{pmatrix} \left[\left[\underbrace{R(\mathbf{x}, T)\, R(\mathbf{y}, P)}_{\text{Turret Transformation}} \right]^{-1} \begin{pmatrix} \tan(H) \\ \tan(V) \\ 1 \end{pmatrix} + \mathbf{t}(z) \right] \tag{A.2}$$

où le point sur le plan rétinien $m \equiv (u, v, 1)^T$ est donné en coordonnées homogènes. La longueur de la focale est contrôlée par le paramètre de zoom z, tel que $f(z) = f_0 + f_1\, z$, étant donné que f s'exprime linéairement en fonction de z. Le translation du centre optique $\mathbf{t}(z)$ s'exprime aussi en fonction de z, mais cette quantité n'a pas été calibrée car non utilisée par les traitements algorithmes.

L'équation (A.2) permet de trouver le pixel correspondant dans l'image où on recherche de l'information. De plus, la surface rétinienne couverte par une cellule de la mosaïque s'écrit

[1] On note $R(\mathbf{u}, \theta)$ une rotation 3D d'angle θ autour de l'axe aligné avec $\mathbf{u}$.

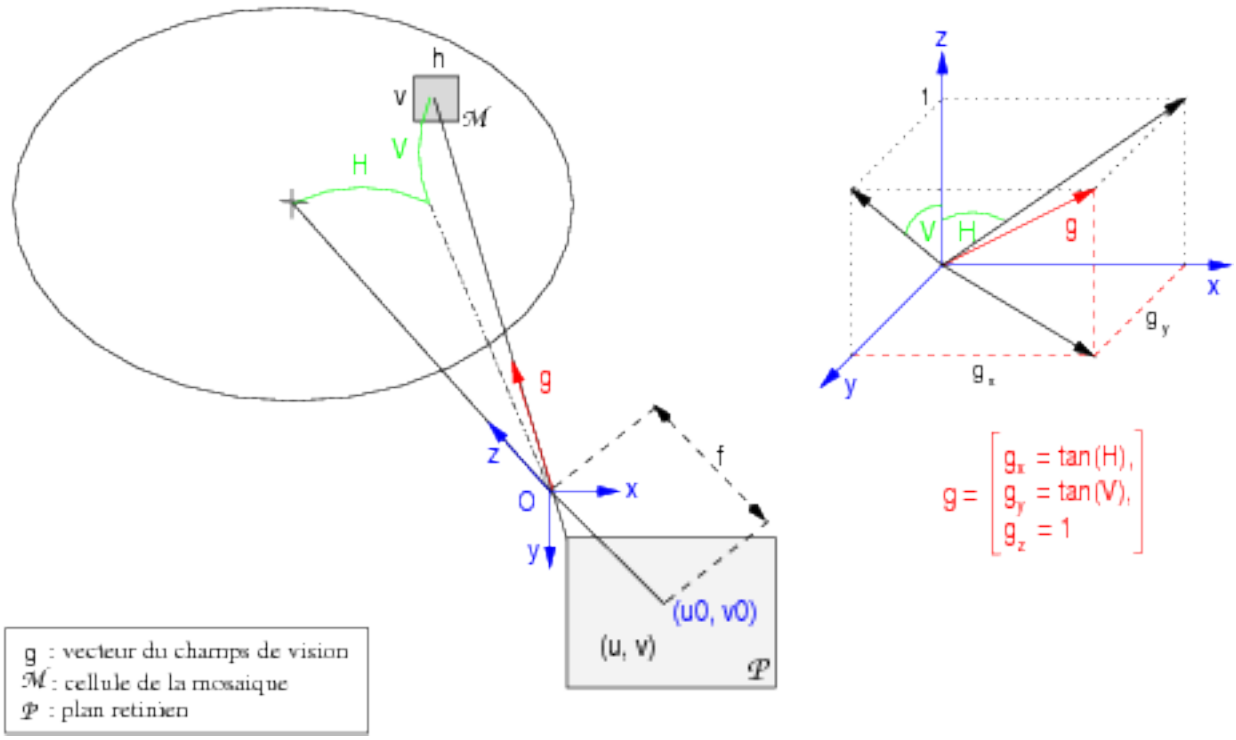


FIG. A.2 – Représentation de la mosaïque. Chaque cellule contient de l'information relative à la scène observée suivant une direction obtenue par projection sur le plan rétinien.

comme :

$$S = \frac{f^2 \cos(H) \cos(V)}{(\cos(V)\cos(T)\cos(P+H) + \sin(T)\sin(V)\cos(H))^3} \quad in \ \ pixel^2 \tag{A.3}$$

On peut donc déterminer automatiquement une *fenêtre de lissage* ce qui va permettre de faire la moyenne des intensités pour chaque cellule.

A.3 Equations d'auto-calibration de la caméra

Pour évaluer efficacement les coefficients de calibration intrinsèque, on peut soit utilisé une rotation pure, comme décrit par [Hartley 97], ou encore utilisé une rotation d'axe fixe, comme démontré par [Viéville 94a]. Etant donné que l'on considère les variations de la longueur de la focale, on ne peut pas se placer dans un cas de rotation pure, en revanche, on peut se placer dans le cas d'une rotation excentré comment décrit dans [Viéville 94a]. Le

déplacement rétinien peut s'exprimer en fonction de l'angle de rotation P par les équations :

$$\left\{ \begin{array}{rcl} u & = & \frac{J_{1,1}\cos(P)+J_{1,2}\sin(P)+J_{1,3}}{J_{3,1}\cos(P)+J_{3,2}\sin(P)+J_{3,3}} \\ v & = & \frac{J_{2,1}\cos(P)+J_{2,2}\sin(P)+J_{2,3}}{J_{3,1}\cos(P)+J_{3,2}\sin(P)+J_{3,3}} \end{array} \right.$$

$$J = \begin{pmatrix} f\,(X-x)+u_0\,(Z-z) & f\,(Z-z)+u_0\,(x-X) & fx+u_0\,z \\ v_0\,(Z-z) & v_0\,(x-X) & fY+v_0\,z \\ Z-z & x-X & z \end{pmatrix} \tag{A.4}$$

où $C = (x, y, z)^T$ désigne le centre de rotation et $M = (X, Y, Z)^T$ est le point 3D dont on cherche la projection.

Avec cette configuration, on peut facilement déduire les coefficients de calibration à partir de **J** :

$$\left\{ \begin{array}{rcl} u_0 & = & \frac{J_{3,1}J_{1,1}+J_{1,2}J_{3,2}}{{J_{3,1}}^2+{J_{3,2}}^2} \\ v_0 & = & \frac{J_{3,1}J_{2,1}+J_{2,2}J_{3,2}}{{J_{3,1}}^2+{J_{3,2}}^2} \\ f & = & \frac{J_{3,1}J_{1,2}-J_{1,1}J_{3,2}}{{J_{3,1}}^2+{J_{3,2}}^2} \\ \epsilon & = & \frac{J_{3,1}J_{2,2}-J_{2,1}J_{3,2}}{{J_{3,1}}^2+{J_{3,2}}^2} \end{array} \right. \tag{A.5}$$

où ϵ est une erreur de mesure (donnée en pixel) qui permet d'évaluer la justesse du modèle. Dans le cas idéal, $\epsilon = 0$, en pratique, on peut utiliser ϵ comme un poids pour moyenner différentes évaluations des coefficients de calibrations, comme cela a été fait dans la partie 4.1.2.

Ces équations sont bien définies si et seulement si

$$d(M, \delta_R) = {J_{3,1}}^2 + {J_{3,2}}^2 = (Z-z)^2 + (X-x)^2 > 0$$

, où δ_R désigne l'axe de rotation ; i.e. dès qu'un point 3D n'est pas sur l'axe de rotation, ce qui est relativement simple à obtenir.

Si on va un peu plus loin, pour chaque point, on peut évaluer la profondeur Z du point 3D. Avec la configuration donnée ci-dessus, on peut déduire une expression simple de Z :

$$Z = \lambda \left[J_{3,3} + \frac{J_{1,2} - u_0\,J_{3,2}}{f} \right]$$

, mais comme le facteur d'échelle monoculaire est indéterminé, on ne peut estimer la la profondeur qu'à un facteur d'échelle λ près.

Annexe B

Approches Réactives Synchrones

B.1 Définitions

Un système *réactif* [Harel 85] est un système informatique qui réagit continûment avec son environnement, à la vitesse déterminée par cet environnement.
Son fonctionnement consiste à émettre des signaux vers l'extérieur comme réaction à l'arrivée d'autres signaux ; le système reste ainsi inactif entre l'occurrence de deux signaux successifs.
Ce type de système s'oppose aux systèmes transformationnels qui consomment leurs entrée en début d'exécution et produisent leur sortie en fin d'exécution. Les systèmes réactifs répondent aux sollicitations de l'environnement durant toute la durée de leur exécution. En l'absence de stimuli extérieurs, le systèmes réactifs sont inertes.

Les caractéristiques d'un système réactif sont en général :

- le déterminisme : au niveau de la spécification fonctionnelle des systèmes réactifs, la même séquence d'entrées produit toujours la même séquence de sortie.
- les contraintes temporelles : imposées par l'environnement car un système réactif doit réagir au rythme de son environnement. Le temps doit être pris en compte :
  1. en terme de durée, en fonction du rythme de la gestion des entrées et de la vitesse de production des sorties en réaction ;
  2. mais aussi en terme d'instants auxquels le système doit déclencher, arrêter l'exécution d'une activité.
- le parallélisme : il est intrinsèque aux systèmes réactifs, car l'environnement et le système évoluent en parallèle.

Classiquement, la programmation logicielle des systèmes réactifs - nous laissons à part la programmation "hardware" - s'est faite suivant deux approches :

1. l'approche déterministe : basée sur les systèmes de transitions tels que les réseaux de Pétri ou les automates.
   L'inconvénient de cette approche est qu'elle ne permet pas d'exprimer explicitement le parallélisme : la gestion du parallélisme est très lourde et est source d'erreurs.
2. l'approche asynchrone : elle permet d'exprimer le parallélisme, c'est le cas des langages de programmation concurrente tels que les langages parallèles ADA, OCCAM.

L'inconvénient de cette approche est de ne pas être déterministe et de rendre la gestion du temps approximative de par l'asynchronisme.

L'approche *synchrone* développée par [Berry 89] permet de réconcilier la programmation déterministe et la programmation concurrente pour les systèmes réactifs.
Cette approche s'appuie sur *l'hypothèse de synchronisme* qui consiste à considérer chaque réaction d'un système comme instantanée. Le système produit alors ses sorties de manière synchrone aux entrées.
Dans cette approche, rien ne se passe dans le système ni l'environnement à l'exception des instants où il y a occurrence de un ou plusieurs événements.

L'approche synchrone est intéressante car elle permet :
– la programmation hiérarchique, concurrente et déterministe ;
– un code efficace et contrôlable ;
– d'utiliser les outils de vérification.

Au niveau mise en oeuvre, l'approche synchrone se trouve à la base de la sémantique de plusieurs langages synchrones de type impératif (ESTEREL, ARGOS) ou encore flots de données (LUSTRE, SIGNAL).

B.2 L'approche flot de données

L'approche flot de données des langages synchrones convient quand la manipulation des données est prédominante comme pour le traitement de signal.
Elle se base sur une description en bloc-diagramme à laquelle on a rajouté une dimension temporelle pour indiquer l'instant pour lequel une entrée ou sortie est disponible.

LUSTRE : ([Caspi 87])

C'est un langage synchrone flot de données développé au laboratoire Verimag [VERIMAG WEB Page]. Un programme LUSTRE se compose de "nodes" fonctionnant en parallèle et activé sur occurrence d'un signal d'entrée. Les "nodes" définissent les équations qui fournissent l'expression du signal de sortie en fonction du signal d'entrée.

Voici comment l'équation $Y_n = Y_{n-1} + (F_n + F_{n-1})/2$ serait écrite en LUSTRE :

```
node integrator(F,STEP,init : real) returns (Y :real);
let
Y = init -> pre(Y) + ((F+ pre(F)) * STEP)/2.0;
tel.
```

Cet exemple montre une partie des opérateurs et données que LUSTRE peut manipuler, à savoir des données de type réelles, entières, booléennes, des constantes ; des opérateurs arithmétiques, booléens (and, or, xor, not), de condition (`if... then... else`) ; et des opérateurs plus spécifiques (`pre` qui fournit la valeur de l'instant précédent, → qui définit la valeur à l'instant 0,...).

De plus, LUSTRE utilise la notion d'horloge booléenne qui va permettre d'activer les "nodes" avec des cadences différentes (l'horloge booléenne sera mise à vrai) :

```
node time_integrator(set,ms :bool; F,STEP,init :real) returns
(Y :real);
var ck : bool;
let
Y = integrator((F,STEP,init) when ck);
ck = true -> set or ms;
tel.
```

Au niveau vérification, si on veut faire des vérifications logiques, un outil de vérification LESAR [Ratel 91] est disponible. Cet outil permet de travailler sur l'automate résultant.
Mais si on s'intéresse plutôt à des propriétés de sûreté du programme, on peut y adjoindre un programme "observateur" qui va recevoir les signaux d'entrée et sortie du programme et qui déclenchera une "alarme" dès que le comportement observé ne satisfait pas une des propriétés. Si il n'y a pas d'alarme, alors le programme respecte les propriétés de sécurité.

SIGNAL : ([Benveniste 92])

C'est un autre langage synchrone flot de données. De la même manière qu'en LUSTRE, un programme SIGNAL est construit sous forme d'équations reliant les signaux. Cependant, contrairement à LUSTRE où un signal représentait une valeur typée à un instant `n`, un signal SIGNAL comprend une valeur typée à laquelle est associée une horloge qui indique les instants où le signal est présent. Donc, pas besoin d'avoir recours à une horloge booléenne extérieure comme en LUSTRE pour spécifier différentes cadences ou synchroniser différents signaux.
De la même manière qu'en LUSTRE, SIGNAL permet de manipuler des données de type entier, booléen, réel ; des opérateurs arithmétiques, et booléens ; plus des opérateurs plus spécifiques au langage.
Comme illustration, reprenons l'exemple donné en B.2 et écrivons le programme SIGNAL correspondant :

```
process integrator =
? real F, STEP, INIT
! real Y;
(| Y := Y$1 + (F + F$1)/2.0 init INIT |)
```

L'exemple B.2 s'écrirait :

```
process time_integrator =
? real F, STEP, INIT
? event set, ms
! real Y;
(| Y ≙ set or ms init true | Y := integratorF,STEP,INIT |)
```

Au niveau vérification, les mêmes remarques faites sur LUSTRE sont applicables à SIGNAL, à ceci près que l'outil de vérification est SIGALI.
SIGALI permet de faire des vérifications concernant la vivacité du système (système n'est jamais en *dead-lock*), l'invariance d'une propriété (on reste toujours avec un ensemble de bons états), l'invariance sous contrôle d'une propriété pour le système (piloter le système en choisissant des événements de manière à rester dans l'ensemble

des bons états), l'atteignabilité d'un ensemble d'états, et l'attractibilité d'un ensemble d'états.
Remarque : SIGALI ne permet de faire des vérifications qu'au niveau de la cohérence temporelle des données.

B.3 L'approche impérative avec ESTEREL

ESTEREL [Boussinot 91], [Berry 92] est un langage de programmation synchrone qui se base sur un modéle réactif, i.e. il suppose que le système est en perpétuel interaction avec son environnement, il interagit avec son environnement par réceptions et émissions de signaux. Le système se comporte comme une boîte noire qui réagit à des événements en entrée et renvoie des événements en sortie. L'hypothèse de synchronisme est forte dans ESTEREL i.e. les sorties sont supposées être fournies de manière absolument synchrone aux entrées, leur calcul "ne prend pas de temps".

ESTEREL possède très peu de possibilités concernant le calcul numérique et les structures de données. Ce type de langage est en effet conçu pour être utilisé en association avec un langage généraliste ([Berry 89]) - en l'occurence le LANGAGE C - et il se décharge donc de ces aspects manipulation de données sur ce dernier.

Propriétés

- La vie du système se décompose en instants. Ces instants correspondent aux moments où le système réagit.
- Hypothèse du Synchronisme Parfait : on suppose en ESTEREL que la diffusion se fait de manière instantannée. Toute activation engendrée par des *input events* est aussitôt suivie de la production des *output events*. C'est comme si les programmes étaient exécutés sur une machine infiniment rapide. Par conséquent, ESTEREL assure :
  - l'atomicité des réactions : pendant que le système réagit, aucune autre activation ne pourra se produire.
- La Diffusion Instantannée : la diffusion est le seul mécanisme de communication dans ESTEREL. Et grâce à son opérateur de parallélisme ||, on peut directement programmer des entités parallèles. Lorsqu'une entité veut communiquer, elle le fait à travers d'un mécanisme de *hand raising*, i.e elle lève la main pour avertir les autres entités qu'elle envoie un signal. **Le signal** est l'objet essentiel de communication de ESTEREL. Les canaux d'entrée et de sortie sont des signaux. Cependant, l'émission du signal est limité aux instants . Les autres entités ne pourront tenir compte du signal émis qu'au même instant où le signal à été émis. Mais, plusieurs signaux d'émissions et de réceptions en séquence peuvent être pris en compte au cours d'un même instant.
- ESTEREL n'admet que des **actions déterministes** : i.e. une suite donnée d'entrées produit toujours la même suite de sorties. ESTEREL permet un parallélisme déterministe.
- Génération d'automates à états finis : en ESTEREL, tout passage vers un programme exécutable se fait à travers une étape intermédiaire de génération d'automates à états finis. ESTEREL assure ainsi le comportement déterministe de l'exécutable.
  L'avantage des automates à états finis :

  - ils sont déterministes, efficaces à l'exécution ;
  - ils peuvent être automatiquement analysés par les systèmes de vérification.

  Leur inconvénient est :
  - ils ne gèrent pas directement le design hiérarchique et les accès concurents, contrairement aux réseaux de Pétri.
- ESTEREL possède très peu de possibilités concernant le calcul numérique et les structures de données.Ce type de langage est en effet conçu pour être utilisé en association avec un langage généraliste ([Berry 89]), et il se décharge donc de ces aspects manipulation de données sur ce dernier.

L'efficacité d'un programme ESTEREL se mesure principalement par la taille de son automate : en l'occurrence le nombre d'appels de transitions et le nombre d'états de l'automate.

Il est important de noter, par ailleurs, qu'il n'y a pas de notion de temps absolu défini en ESTEREL. Le temps est simplement considéré comme un signal parmi d'autres.

Exemple Voici un exemple de programme ESTEREL :

```
module FOO :
input READY, MS, STOP;
output GO_ON, GO_OFF, RED_ON, RING_BELL;
every READY do emit RING_BELL end
||
do await 3 MS watching STOP timeout exit ERROR end; emit GO_ON;
   handle ERROR do
emit RED_ON; emit GO_OFF
end;
end.
```

Cet exemple présente une partie des opérateurs du langage ESTEREL dont le `module` est l'unité de programmation de base. Il se compose :

- d'une partie déclaration des signaux (`input`, `output`) qui composent l'interface du module, mais on peut aussi déclarer des types, des constantes, des procédures ou des fonctions.
- d'une partie instructions utilisant des opérateurs de condition (`if ... then ... else ... end`), d'itération (`loop ... end`), d'affectation (`:=`), de parallélisme (`... || ...`) ; des opérateurs qui manipulent les signaux tels que `emit` , `await`, ou encore le *chien de garde* `do ... watching ... timeout`, ou enfin un opérateur qui gère les exceptions `trap` etiquette `in <instructions> end handle` etiquette `... do ... end`.

Annexe C

Introduction au langage ESTEREL

Cette annexe présente les principales instructions du langage ESTEREL dont une description plus exhaustive de ce langage est disponible dans [Berry 92].

1. la structure d'un programme ESTEREL
   Syntaxe : `module` *nom_du_module* :
   `input`
   SigInput1, SigInput2, ...;
   `output`
   SigOutput1, SigOutput2, ...;
   [`signal`
   SigLocal1,SigLocal2, ... in]
   [`var`
   X1 : *type_varX1*, *X2* : *type_varX2*, ... `in`]
   corps
   `end`
   .
   `input` est la zone de déclaration des signaux d'entrée du module, `output` est la zone de déclaration des signaux de sortie.
   corps représente le corps du module.
   `signal` introduit la déclaration de signaux locaux à un module. `var` introduit la déclaration des variables locales au module.
2. les commentaires
   Syntaxe : `%` *commentaires*
3. l'instruction `nothing`
   Syntaxe : `nothing;`
   C'est l'instruction vide.
4. l'instruction `emit` *S*
   Syntaxe : `emit` *S*
   `emit` *S* permet d'émettre un signal *S*.
5. l'instruction `?SIG`
   Syntaxe : `?SIG`
   SIG un signal valué déclaré comme *SIG(integer)*. `?SIG` renvoie la valeur du signal *SIG*.

6. la primitive `;`
   Syntaxe : *instr1* `;`
   instr2 `;`
   ...
   Cette primitive traduit le mise en séquence d'instructions.

7. l'instruction ||
   Syntaxe : `instr1;`
   ||
   `instr2;`
   || représente l'opérateur parallèle. Les branches *instr1* et *instr2* sont démarrées en même temps, et l'instruction parallèle se termine lorsque les deux branches sont terminées.

8. la structure `do - upto`
   Syntaxe : `do`
   instr
   `upto Occ`
   C'est l'instruction de réception de signal. Elle ne se termine que sur réception de l'événement *Occ*(le signal *Occ* est défini à partir de l'instant présent) et non si le corps *instr* de la boucle se termine. Si *Occ* arrive à l'instant présent, *instr* ne sera pas exécuté. Sinon, *instr* se termine mais l'instruction ne s'achèvera qu'à la réception de *Occ*.

9. l'instruction `await`
   Syntaxe : `await S`
   Cette instruction permet d'attendre l'occurrence du signal *S* à partir de l'instant futur. Cette instruction est en fait équivalente à : `do nothing upto S`.

10. l'instruction `await immediate`
    Syntaxe : `await immediate S`
    `immediate` permet de spécifier que l'on veut que le système réagisse dans l'instant présent et non dans l'instant d'après. `await immediate S` permet d'attendre une occurrence de *S* dans l'instant présent ou futur.

11. la structure `await - case`
    Syntaxe : `await`
    `case S1 do`
    body1
    `case S2...`
    `watching S`
    Cette structure représente la sélection d'événements ou attente multiple. La première des occurrences *Si* satisfaite détermine l'action à effectuer et qui est lancée instantanément. Cependant, si plusieurs occurrences sont satisfaites simultanément, seule la première dans la liste est lancée.

12. la structure `present - if - then - else - end`

Syntaxe : `present` *S* `then`
body1
`else`
body2
`end`
Ceci permet de tester la présence d'un signal *S*.

13. l'instruction `do - watching - timeout` x
Syntaxe : `do`
instr1
`watching` *Occ*
`timeout` *instr2* `end`
Cette instruction permet d'implanter le système du chien de garde. Si *instr1* se termine avant l'occurrence *Occ*, le chien de garde se termine avec *instr1*. Sinon, dès l'arrivée de *Occ*, *instr1* est tuée et *instr2* est lancée.

14. la primitive `loop - each` x
Syntaxe : `loop`
corps
`each` *S*
Cette primitive permet de traiter le déclenchement répétitif du signal *S*. Cette structure permet, par exemple, de rendre compte de l'état de ressources. Avec ce type de boucle, le *corps* est toujours exécuté au moins une fois. Et à chaque réception du signal *S*, le corps de la boucle est réexécuté.

15. la construction `trap - in - exit - ||  - exit - end`
Syntaxe : `trap` *except*
body1
`exit` *except*
||
body2
`exit` *except*
Cette construction constitue un puissant mécanisme de sortie de bloc. `trap` *except* définit un bloc dont on sort instantanément quand `exit` *except* est exécuté.
body1 et *body2* sont exécutés et se terminent normalement si *body1* et *body2* ne génèrent pas *except*. Sinon, `trap` permet de sortir du bloc à la réception de *except*.

16. l'instruction `copymodule` x
Syntaxe : `copymodule nom_de_module`
Cette instruction permet de programmer de manière modulaire sous ESTEREL. `copymodule` permet de copier un module écrit à un endroit à l'intérieur d'un autre module ($\sim$ appel d'un module à l'intérieur d'un autre).

Bibliographie

[Aloimonos 87] J.Y Aloimonos, I. Weiss & A. Bandyopadhyay. *Active Vision*. In Proceedings of the International Conference on Computer Vision, pages 35–54, London, June 1987.

[Aloimonos 90] J.Y Aloimonos. *Purposive and Qualitative Active Vision*. In ICPR, Atlantic City, New Jersey, pages 346–360, June 1990.

[ARGES 99] ARGES. *The* ARGES *System* WEB *Page*. "http ://www-sop.inria.fr/members/Thierry.Vieville/Demos/Arges/", 1999. Robotvis Team ; INRIA.

[Astraudo 92] C. Astraudo & J.J. Borrelly. *Simulation of Multiprocessor Robot Controllers*. In IEEE International Conference on Robotics and Automation, pages 573–578, Nice, France, May 1992.

[Backus 60] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden & Michael Woodger. *Report on the Algorithmic Language ALGOL 60*. Communications of the ACM, vol. 3, no. 5, pages 299–314, May 1960.

[Bajscy 88] R. Bajscy. *Active Perception*. Proceedings of the IEEE, Special Issue on Computer Vision, vol. 76, no. 8, pages 996–1005, August 1988.

[Benveniste 92] A. Benveniste, M. Le Borgne & P. Le Guernic. SIGNAL *as a Model for Real-Time and Hybrid Systems*. Rapport technique RR-1608, INRIA, February 1992.

[Berry 89] G. Berry. *Real-Time Programming : Special Purpose or General Purpose languages*. In Information Processing 89 – IFIP 89, pages 11–17. G.X. Ritter (ed), Elsevier Science Publishers, B.V. North Holland, September 1989.

[Berry 91] G. Berry & G. Gonthier. The ESTEREL synchronous programming, language : Design, semantics, implementation, May 1991.

[Berry 92] G. Berry & G. Gonthier. *The Synchronous Esterel Programming Language : Design, Semantics, Implementation*. Science of Computer Programming, vol. 19, no. 2, pages 87–152, November 1992.

[Berry 96] G. Berry. The constructive semantics of ESTEREL. available at "http ://www.inria.fr/meije/esterel/esterel-eng.html", draft book edition, 1996.

[Borras 88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang & V. Pascual. *Centaur : the system*. In Third Symposium on Software Development Environments (SDE3), ACM SIGSOFT'88, Boston, USA, December 1988.

[Borrelly 90] J.J Borrelly & D. Simon. *Proposition d'Architecture de Contrôleur Ouvert pour la Robotique*. Rapport de Recherche RR-1304, INRIA, October 1990.

[Bouali 96] A. Bouali, A. Ressouche, V. Roy & R. de Simone. *The FC2Tools set*. LNCS 1102, Computer Aided Verification, 1996. Fc2Tool Web Page :"http ://www.inria.fr/meije/verification/".

[Boussinot 91] F. Boussinot & R. de Simone. *The* ESTEREL *Language*. Proceedings of the IEEE, Special Issue, vol. 79, no. 9, September 1991.

[Bremond 97] F. Bremond. *Environnement de résolution de problèmes pour l'interprétation de séquences d'images*. PhD thesis, Université Nice-Sophia Antipolis, October 1997.

[Brooks 86] R. A. Brooks. *A Robust Layered Control System for Mobile Robot*. IEEE Journal of Robotics and Automation, vol. 2, no. 1, pages 14–23, March 1986.

[Brown 93] Christopher Brown, David Coombs & John. Soong. *Real-Time Smooth Pursuit Tracking*. In Andrew Blake & Alan Yuille, editeurs, Active Vision, chapitre VIII, pages 123–136. The MIT Press, 1993.

[Caspi 87] P. Caspi, D. Pilaud, N. Halbwachs & J. Plaice. LUSTRE *: a declarative language for programming synchronous systems*. In 14th ACM Symposium on Principles of Programming Languages, München, Germany, January 1987.

[Caspi 94] P. Caspi, A. Girault & D. Pilaud. *Distributing reactive systems*. In 7th International Conference on Parallel and Distributed Computing Systems, Las Vegas, USA, October 1994.

[Chleq 99] N. Chleq, F. Brémond & M. Thonnat. *Image Understanding for Prevention of Vandalism in Metro Stations*. In C. S. Regazzoni, G. Fabri & G. Vernazza, editeurs, The Kluwer International Series in Engineering and Computer Science, pages 106–116. Kluwer Academic Publishers, 1999.

[Christensen 97] H.I. Christensen & W. Förstner. *Performance Characteristics of Vision Algorithms*. Machine Vision and Applications, vol. 9, pages 215–218, 1997.

[Clément 91] D. Clément, V. Prunet & F. Montagnac. *Integrated Software Components : A Paradigm for Control Integration*. In E. Albert & W. Herbert, editeurs, Proceedings of Software Development Environments and CASE Technology, volume 509 of *LNCS*, pages 167–177. Springer Verlag, June 1991.

[Clément 93] V. Clément, M. Thonnat & J. Van den Elst. *Supervision of the Perception Tasks for Autonomous Systems : the* OCAPI *approach*. Rapport technique RR-2000, INRIA, June 1993.

[Coste-Manière 97] E. Coste-Manière & N. Turro. *The* MAESTRO *Language and its Environment : Specification, Validation and Control of Robotic Missions*. In IEEE/RSJ Intl Conf on Intelligent Robots and Systems, IROS'97, volume 2, pages 836–841, Grenoble, France, September 1997. Maestro web page :"http ://www.inria.fr/icare/maestro/".

[Crowley 94] J. L. Crowley & H. Christensen. *Vision as Process : Integration and Control of Real Time Active Vision System*. Experimental Environments for Computer Vision and Image Processing, vol. 11, pages 127–155, 1994.

[Crubezy 99] M. Crubezy. *Pilotage de programmes pour le traitement d'images médicales*. PhD thesis, Université de Nice-Sophia Antipolis, February 1999.

[Crubézy 97] M. Crubézy, F. Aubry, S. Moisan, V. Chameroy, M. Thonat & R. Di Paola. *Managing complex processing of medical image sequences by program supervision techniques*. In Proceedings of the SPIE international symposium Medical Imaging 1997 (MI'97), pages 614–625, Newport Beach, California, USA., February 1997.

[Davis 97] J. W. Davis & A. Bobick. *The Representation and Recognition of Human Movement using Temporal Templates*. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pages 928–934, San Juan, Puerto Rico, June 1997.

[den Elst 96] J. Van den Elst. *Modélisation de connaissances pour le pilotage de programmes de traitement d'images*. PhD thesis, Université de Nice-Sophia Antipolis : sciences pour l'ingénieur, October 1996.

[Deriche 87a] R. Deriche. *Using Canny's Criteria to Derive a Recursively Implemented Optimal Edge Detector*. The International Journal of Computer Vision, vol. 1, no. 2, pages 167–187, May 1987.

[Deriche 87b] R. Deriche. *Using Canny's Criteria to Derive a Recursively Implemented Optimal Edge Detector*. International Journal of Computer Vision, vol. 1, no. 2, pages 167–187, May 1987.

[Förstner 96] W. Förstner. *10 Pros ans Cons Against Performance Characterization of Vision Algorithms*. In Workshop on Performance Characteristics of Vision Algorithms, Cambridge, UK, April 1996.

[Gal 98] C. Le Gal. *Intégration d'une couche décisionnelle à* ORCCAD. In 10èmes Journées des Jeunes Chercheurs en Robotique, Amiens, France, 1998.

[Gal 99] C. Le Gal, J. Martin & G. Durand. SMARTOFFICE *: An Intelligent and Interactive Environment*. In to appear in First International Workshop on Managing Interactions in Smart Environments, Dublin, Ireland, 1999.

[Gallmeister 95] B. O. Gallmeister. POSIX.4, programming for the real world. O'Reilly & Associates, inc., 1st edition, January 1995.

[Gamma 95] E. Gamma, R. Helm, R. Johnson & J. Vlissides. Design patterns. Addison Wesley, 1995.

[Gaudel 96] M.C. Gaudel, B. Marre, F. Schlienger & G. Bernot. Précis de génie logiciel. Masson, first edition, May 1996.

[Gillies 95] D. W. Gillies & J. W.-S. Liu. *Scheduling Tasks with AND/OR Precedence Constraints*. SIAM Journal on Computing, vol. 24, no. 4, pages 797–810, August 1995.

[Ginhac 98] D. Ginhac, J. Sérot & J.P. Dérutin. *Fast prototyping of image processing applications using functional skeletons on MIMD-DM architecture*. In IAPR Workshop on Machine Vision Applications, Chiba, Japan, November 1998.

[Giraudon 96] G. Giraudon & F. Sandakly. *Interprétation de scènes d'intérieur pour un robot mobile*. Techniques et science informatiques, vol. 15, no. 6, pages 1–31, 1996.

[Guernic 91] P. Le Guernic, T. Gautier, M.. Le Borgne & C. Le Maire. *Programming Real-Time Applications with Signal*. Proceedings of the IEEE, vol. 79, no. 9, pages 1321–1336, 1991.

[Gusev 98] S. V. Gusev, I. A. Makarov, I. E. Paromtchik, V. A. Yakubovich & C. Laugier. *Adaptive motion control of a nonholonomic vehicle*. In Proc. of the IEEE Int. Conf. on Robotics and Automation, volume 4, pages 3285–3290, Leuven (BE), May 1998.

[Hager 96] G. D. Hager & K. Toyama. *XVision : combining image warping and geometric constraints for fast visual tracking*. Lecture Notes in Computer Science, Springer Verlag, vol. 1065, 1996.

[Hager 98] G. D. Hager & K. Toyama. *X Vision : A Portable Substrate for Real-Time Vision Applications*. Computer Vision and Image Understanding : CVIU, vol. 69, no. 1, 1998.

[Harel 85] D. Harel & A. Pnueli. *On the Development of Reactive Systems, Logic and Models of Concurrent Systems*. Springer-Verlag, pages 477–498, 1985.

[Hartley 97] Richard Hartley. *Self-Calibration of Stationary Cameras*. The International Journal of Computer Vision, vol. 22, no. 1, pages 5–24, February 1997.

[Ikeuchi 96] K. Ikeuchi & M. Hebert. *Task-Oriented Vision*. In Exploratory Vision : The Active Eye, Springer Series in Perception Engineering, March 1996.

[Kapellos 94] K. Kapellos. *Environnement de programmation des applications robotiques reactives*. PhD thesis, Ecole des Mines de Paris : informatique temps reel, robotique, automatique, Novembre 1994.

[Kapellos 97] K. Kapellos, D. Simon, S. Granier & V. Rigaud. *Distributed Control of a Free-floating Underwater Manipulation System*. In International Symposium on Experimental Robotics, Barcelona, Spain, June 1997.

[Khatib 96] O. Khatib, K. Yokoi, K. Chang, D. C. Ruspini, R. Holmberg & A. Casal. *Coordination and Decentralized Cooperation of Multiple Mobile Manipulators*. Journal of Robotic Systems, vol. 13, no. 11, pages 755–764, 1996.

[Lacan 98] P. Lacan, J.N. Monfort, L.V.Q. Ribal, A. Deutsch & G. Gonthier. *ARIANE 5, the Software Reliability Verification Process : The ARIANE 5 Example*. Proceedings DASIA'98, Data Systems In Aerospace, ESA Publications, no. SP-422, May 1998.

[Lavarenne 91a] C. Lavarenne, O. Seghrouchni, Y. Sorel & M. Sorine. *The* SYNDEX *Software Environment for Real-Time Distributed Systems Design and Implementation*. In European Control Conference, 1991.

[Lavarenne 91b] C. Lavarenne & Y. Sorel. SYNDEX*, un environnement de programmation pour applications de traitement du signal distribuées*. In Actes du treizième Colloque GRETSI, September 1991.

[Luotonen 94] A. Luotonen & K. Altis. *World-Wide Web Proxies*. Computer Networks and ISDN Systems, vol. 27, no. 2, pages 147–154, 1994.

[Lux 97] A. Lux & B. Zoppis. *An Experimental Multi-Language Environment for the Development of Intelligent Robot Systems*. In Proceedings of the Fourth International Symposium on Intelligent Robotic Systems, 1997.

[Marcos 95] M. Marcos, S. Moisan & A. P. del Pobil. *Verification and validation of knowledge-based program supervision systems*. In I3E International Conference on Systems, Man and Cybernetics (SMC'95), Vancouver, Canada, October 1995.

[Marcos 97] M. Marcos, S. Moisan & A. P. del Pobil. *A Model-Based Approach to the Verification of Program Supervision Systems*. In Proceedings of 4th European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV'97), pages 231–241, Leuven, Belgium, June 1997.

[Meyer 88] B. Meyer. Object-oriented software construction. Prentice Hall, first edition, 1988.

[Meyer 90] B. Meyer. Introduction to the theory of programming languages. International Series in Computer Science. Prentice-Hall, 1990.

[Mundy 93] J. Mundy, T. Binford, T. Boult, A. Hanson, R. Beveridge, R. Haralick, V. Ramesh, C. Kohl, D. Lawton, D. Morgan, K. Price & T. Strat. *The Image Understanding Environment Program*. In Proceedings IEEE CVPR'93, pages 406–416, June 1993.

[Murcia 97] C. De Murcia, M. Niemaz & T. Viéville. *Détection et suivi de cibles sur une durée indéterminée*. In Proceedings of Orasis'97, October, La Colle s/Loup, France, 1997.

[Murray 93] D.W. Murray, P.F. MacLauchlan, I.D. Reid & P.M. Sharkey. *Reactions to peripheral image motion using a head/eye platform*. In 4th ICCV, pages 403–411. IEEE Society, 1993.

[Murray 94] D. Murray & A. Basu. *Motion Tracking with an Active Camera*. IEEE Transaction Pattern Analysis and Machine Intelligence, vol. 16, no. 5, pages 449–459, 1994.

[Najjar 98] W. Najjar, B. Draper, A.P.W. Böhm & R. Beveridge. *The* CAMERON *Project : High-Level Programming of Image Processing Applications on Reconfigurable Computing Machines*. In Proceedings of the Workshop on Reconfigurable Computing, PACT'98, pages –, Paris, France, 1998.

[Nordlund 96] P. Nordlund & T. Uhlin. *Closing the Loop : Detection and Pursuit of a Moving Object by a Moving Observer*. Image and Vision Computing, vol. 14, pages 265–275, May 1996.

[Pinhanez 99] C. Pinhanez & A. Bobick. *Using Computer Vision to Control a Reactive Computer Graphics Character in a Theater Play*. In H.I. Christensen, editeur, Lecture Notes in Computer Science, volume 1542, pages 66–82. Springer Verlag, January 1999.

[Pissard-Gibollet 95] R. Pissard-Gibollet, K. Kapellos, P. Rives & J.J. Borrelly. *Real-Time Programming of Mobile Robot Actions Using Advanced Control Techniques*. In Fourth International Symposium on Experimental Robotics, ISER95, Stanford, USA, June 1995.

[Ratel 91] C. Ratel, N. Halbwachs & P. Raymond. *Programming and verifying critical systems by means of the synchronous data-flow programming language* LUSTRE. In ACM-SIGSOFT'91 Conference on Software for Critical Systems, New Orleans, USA, December 1991.

[Robert 97] Paul Robert. Le nouveau petit robert. Dictionnaires Le Robert, 1997.

[Rosenblatt 97] J. K. Rosenblatt. *A distributed Architecture for Mobile Navigation*. PhD thesis, Robotic Institute, Carnegie Mellon University, January 1997.

[Roy 90] V. Roy & R. de Simone. *Auto and Autograph*. In R. Kurshan, editeur, Workshop on Computer Aided Verification, New Brunswick, USA, June 1990.

[Ruspini 98] D. C. Ruspini & O. Khatib. *Dynamic Models for Haptic Rendering Systems*. In Proc. of the International Symposium on Advances in Robot Kinematics, pages 523–532, Strobl/Salzburg, Austria, June 1998.

[Rutten 95] E. Rutten & F. Martinez. SIGNAL GTI *: implementing task preemption and time intervals in the synchronous data flow language* SIGNAL. In 7th Euromicro Workshop on Real Time Systems, pages 1–31, Odense, Danemark, June 1995.

[BIP WEB Page] BIP WEB Page. *"http ://www.inrialpes.fr/bip/"*.

[CHIR WEB Page] CHIR WEB Page. *"http ://www-sop.inria.fr/chir/"*.

[FC2TOOL WEB Page] FC2TOOL WEB Page. *"http ://www.inria.fr/meije/verification/"*.

[ICARE WEB Page] ICARE WEB Page. *"http ://www.inria.fr/icare/"*.

[ORCCAD WEB Page] ORCCAD WEB Page. *"http ://sed.inrialpes.fr/Orccad/"*.

[ROBOTVIS WEB Page] ROBOTVIS WEB Page. *"http ://www.inria.fr/en/teams/robotvis/"*.

[SOPHTALK WEB Page] SOPHTALK WEB Page. *"http ://www.inria.fr/croap/sophtalk/sophtalk.html"*.

[TARGET WEB Page] TARGET WEB Page. *"http ://www.targetjr.org/"*.

[TOLÈRE WEB Page] TOLÈRE WEB Page. *"http ://www.inrialpes.fr/bip/people/girault/Projets/Incitative/index.html"*.

[VERIMAG WEB Page] VERIMAG WEB Page. *"http ://www-verimag.imag.fr/"*.

[Schneider 97] W. Schneider, W. Eckstein & C. Steger. *Real-Time Visualization of the Interactive Parameters Changes in Image Processing Systems*. In Proceedings SPIE conf. 3017, Visual Data Exploration and Analysis IV, pages 286–295, San Diego, February 1997.

[Serot 99] J. Serot, D. Ginhac & J.P. Derutin. *SKIPPER : A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications*. Lecture Notes in Computer Science, vol. 1662, 1999.

[Smith 97] S.M. Smith & J.M. Brady. *SUSAN - a new approach to low level image processing*. IJCV, vol. 23, no. 1, pages 45–78, 1997.

[Team 98] The Orccad Team. *An Integrated and Modular Approach for the Specification, the Validation and the Implementation of Complex Robotics Missions*. International Journal of Robotics Research, vol. 17, no. 4, pages 338–359, April 1998. Special Issue on Integreated Architectures for Robot Control and Programming.

[Thonnat 94] M. Thonnat, V. Clément & J. Van den Elst. *Supervision of perception tasks for autonomous systems : the OCAPI approach*. Journal of Information Science and Technology, vol. 3, no. 2, pages 140–162, 1994.

[Turro 99] Nicolas Turro. MAESTRO *: une approche formelle pour la programmation d'applications robotiques*. PhD thesis, Université de Nice-Sophia Antipolis, September 1999.

[Umbaugh 98] S. Umbaugh. Computer vision and image processing : a practical approach using CVIPTOOLS. Prentice Hall, first edition, 1998.

[Viéville 94a] T. Viéville. *Autocalibration of Visual Sensor Parameters on a Robotic Head*. Image and Vision Computing, vol. 12, 1994.

[Viéville 94b] Thierry Viéville & Olivier D. Faugeras. *Robust and Fast Computation of Edge Characteristics in Image Sequences*. The International Journal of Computer Vision, vol. 13, no. 2, pages 153–179, October 1994.

[Viéville 95] T. Viéville, E. Clergue, R. Enciso & H. Mathieu. *Experimentating with 3-D vision on a robotic head*. Robotics and Autonomous Systems, 1995. 14(1).

[Vincent 97] R. Vincent, M. Thonat & JC. Ossola. *Program Supervision for Automatic Galaxy Classification*. In International Conference on Imaging Science, Systems and Technology (CISST'97), Las Vegas, USA, June 1997.

[Weems 91] C.C. Weems, E. M. Riseman, A. R. Hanson & A. Rosenfeld. *The* DARPA *Image understanding Benchmark for Parallel Computers*. Journal of Parallel and Distributed Computing, vol. 11, no. 1, pages 1–24, 1991.

[Willson 94] Reg Willson. *Modeling and Calibration of Automated Zoom Lenses*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1994.

[Zimmerman 80] H. Zimmerman. OSI *Reference Model - The* ISO *Model of Architecture for Open Systems Interconnection*. IEEE Transaction on Communications, vol. 28, no. 4, April 1980.

[Zoppis 97] B. Zoppis. *Outils pour l'intégration et le contrôle en vision et robotique mobile*. PhD thesis, Institut National Polytechnique de Grenoble, June 1997.

Glossaire

analyse statique : extraction automatique de propriétés sur des programmes informatiques. Ces propriétés sont une description formelle possible du comportement du programme.

architecture client-serveur : architecture de réseau dans laquelle les traitements sont répartis entre les clients qui demandent les informations dont ils ont besoin au(x) serveur(s).

bloc fonctionnel(block-diagram) : unité d'organisation de programme, instanciable, retournant un ou plusieurs éléments de données après son exécution.

broadcast : (ou diffusion générale) diffusion simultanée d'un message à toutes les machines présentes sur un réseau.

cadence : représente la durée qui s'écoule entre deux données d'entrée dans un système.

communication point à point : caractérise une communication dans laquelle on ne s'adresse qu'à un seul destinataire.

compatibilité : [Meyer 88]

La compatibilité est l'aptitude des logiciels à pouvoir être combinés les uns avec les autres. Les logiciels doivent interagir les uns avec les autres. Donc pour que les programmes puissent exploiter les résultats d'autres programmes, il faut une conception homogène et une utilisation de conventions normalisées pour la communication entre programmes. Les solutions possibles comprennent :

- des formats de fichiers normalisés. E.g. en UNIX, chaque fichier de texte est une suite de caractères.
- des structures de données normalisées. E.g. en LISP, les données et les programmes sont représentées par des arbres binaires.
- des interfaces utilisateur normalisées. E.g. en SMALLTALK les outils appliquent les mêmes principes pour la communication avec l'utilisateur : fenêtre, icône, graphique, etc.
- plus généralement, on pourrait définir des protocoles d'accès standard pour toutes les entités importantes manipulées par un système.

Design Patterns :

mécanisme qui facilite la réutilisation large de la conception logicielle (Software Design) et par conséquent son implémentation en fournissant les structures de réutilisabilité au niveau de la conception proprement dite. Ce mécanisme est un élément phare de la conception orientée-objet. Un "design pattern" est défini par un certain nombre de classes abstraites qui sont spécialisées relativement à un problème donné à résoudre.

efficacité : [Meyer 88]
L'efficacité est la bonne utilisation des ressources du matériel.
Le matériel regroupe les processeurs, les mémoires internes et externes, les matériels de communication, etc.

e.g. : voir **exempli gratia**

ESTEREL : [Berry 92]
Langage synchrone textuel, de style impératif adapté à la programmation du contrôle des systèmes réactifs.

etc. :
Abréviation latine de *et ceatera*, qui signifie et ainsi de suite.

exempli gratia :
Expression latine qui signifie par exemple. On la trouve souvent en abrégé notée *e.g.*.

extensibilité : [Meyer 88]
L'extensibilité est la facilité d'un logiciel aux changements de spécification.
Deux principes essentiels pour améliorer l'extensibilité :

la simplicité de conception : une architecture simple est plus facile à modifier qu'une architecture complexe.

la décentralisation : plus les modules d'une architecture logicielle sont autonomes, plus il est probable qu'une modification simple n'affectera qu'un seul module, ou un nombre restreint de modules, plutôt que de déclencher une réaction en chaîne sur tout le système.

facilité d'utilisation : [Meyer 88]
La facilité d'utilisation est la facilité avec laquelle les utilisateurs d'un logiciel peuvent apprendre comment l'utiliser, comment le faire fonctionner, comment préparer les données, mais aussi comment interpréter les résultats et réparer les effets en cas d'erreur.

fiabilité : [Meyer 88]
La fiabilité est un concept qui couvre à la fois la validité et la robustesse.

haptic : système à retour d'effort qui permet de toucher et manipuler des objets dans un environnement virtuel.

id est :
Expression latine qui signifie c'est-à-dire. On la trouve souvent en abrégé notée *i.e.*.

intégration : [Gaudel 96]
L'intégration consiste à assembler tout ou partie des composants d'un logiciel pour obtenir un système exécutable.

intégrité : [Meyer 88]
L'intégrité est l'aptitude des logiciels à protéger leurs différentes composantes (programmes, données, documents) contre des accès ou des modifications non autorisés.

latence : représente la durée d'exécution d'une itération de l'application conduisant à produire le prochain résultat en sortie, en réponse à une donnée d'entrée.

langage synchrone : langage qui suppose que la machine qui l'exécute soit infiniment rapide par rapport à la dynamique du système commandé, et produise ses sorties au "même instant" que ses entrées. Cette classe de langage est généralement basé sur une sémantique rigoureuse, et comprend les langages de programmation de flot de données (LUSTRE et SIGNAL), des langages de contrôle textuels (ESTEREL) ou graphiques

(ARGOS, STATECHARTS ou SYNCHCHARTS).

LUSTRE : [Caspi 87]
Langage synchrone, de style déclaratif adapté à la programmation du flot de données des systèmes réactifs.

MAESTRO : [Turro 99]
Langage de programmation dédié au contrôle robotique, qui se traduit en langage ESTEREL.

MIMD : Multiple Instruction Multiple Data. Plusieurs données traitées en même temps par plusieurs instructions. Cette techniques est utilisée dans les ordinateurs parallèles.

multicast : (ou diffusion restreinte, multipoint) diffusion simultanée d'un message en direction d'un sous-ensemble de machines sur un réseau.

portabilité : [Meyer 88]
La portabilité est la facilité avec laquelle un produit logiciel peut être adapté à différents environnements matériels et logiciels.

processus : Un processus est l'activité résultant de l'exécution d'un programme séquentiel, avec ses données, par un processeur.

processus léger : voir **thread**.

programmation déclarative : En programmation déclarative, une application est programmée comme un ensemble de déclarations qui décrivent cette dernière. Ces déclarations sont enregistrées et compilées sous la forme d'une base de connaissances. Les résultats souhaités sont obtenus par consultation de la base de connaissances. À priori, il s'agit de styles de programmation de haut niveau, où l'utilisateur n'a pas à se préoccuper de la suite de transformations élémentaires qui font passer des données d'une application à ses résultats.

programmation impérative : En programmation impérative, une application est programmée comme une suite de transformations qui fait passer de l'état initial du système que l'on veut modéliser à son état final. Ces transformations peuvent porter sur des variables élémentaires, des structures de données, voire des objets plus complexes.

protocole : [Zimmerman 80]
Un ensemble de règles et de formats qui régissent les communications d'entités paires (ou peers).

proxy : mot de langage anglaise, qui désigne une personne ou une organisation qui a toute autorité d'agir au nom d'un tiers (traduction française courante : mandataire).

queue de message :
Une queue de messages est un objet qui permet le passage d'un nombre arbitraire de données entre des processus d'une application.

réutilisabilité : [Meyer 88]
La réutilisabilité est l'aptitude d'un logiciel à être réutilisé en tout ou en partie pour de nouvelles applications.
Ce besoin de réutilisabilité vient de ce que beaucoup d'éléments des logiciels se ressemblent. Donc il est logique de vouloir exploiter leurs points communs et d'éviter d'avoir à réinventer des solutions à des problèmes déjà rencontrés.
Résoudre le problème de réutilisabilité signifie essentiellement que l'on pourra réécrire moins de logiciels et donc consacrer plus d'efforts à améliorer d'autres facteurs comme la validité, la robustesse, etc.

robustesse : [Meyer 88]
La robustesse est l'aptitude d'un logiciel à fonctionner même dans des conditions anormales.
L'enjeu est le comportement dans des circonstances non couvertes par les spécifications. La robustesse, c'est l'assurance que dans des cas extrêmes, le logiciel ne va pas déclencher une catastrophe ; il doit annuler son exécution proprement ou se mettre dans un état de "dégradation harmonieuse".

sémantique : La sémantique d'un langage est un mécanisme qui permet de donner un *sens* à chaque programme écrit avec ce langage. La notion de *sens* regroupe ici à la fois la notion de correction (validité) de programmes et d'évaluation (exécution) des programmes.

sémantique opérationnelle : Elle décrit comment un programme interagit effectivement avec son environnement. Elle favorise l'implantation d'un interpréteur du langage.

sémaphore : Un sémaphore est un objet qui possède une valeur entière et un ensemble de processus bloqués associés à lui :
si $sem >= 0$ aucun processus n'est bloqué sur le sémaphore ;
si $sem < 0$ $|sem|$ = le nombre de processus bloqués sur le sémaphore
En fonction de la valeur de sem, on distingue : *le sémaphore binaire* pour $sem \in [0, 1]$ et *le sémaphore à compte* pour $sem \in I\!N$.
Le sémaphore binaire sert comme mécanisme élémentaire de synchronisation entre 2 processus ou pour protéger une ressource partagée par plusieurs processus.
Le sémaphore à compte permet de garder la trace de la disponibilité des ressources, en mémorisant le nombre d'appel qu'il a reçu pour une ressource donnée.

SIMD : Single Instruction Multiple Data. Plusieurs données traitées par une seule instruction. Cette technique est utilisée par les ordinateurs vectoriels.

SIGNAL :[Guernic 91]
Langage synchrone, de style déclaratif adapté à la programmation du flot de données des systèmes réactifs.

socket : interface utilisée pour les échanges de données locaux ou distants à travers le réseau.

spécification logicielle : [Meyer 90]
l'action de définir avec précision le but de chaque programme ou de chaque élément de programme sans privilégier une implantation particulière.

système : un ensemble "d'activités" correspondant à un ou plusieurs traitements effectués en séquence ou en concurrence et qui communiquent éventuellement entre eux.

système de pilotage de programme : [Crubezy 99]
l'automatisation de ou partie du raisonnement relatif à l'utilisation d'un ensemble de programmes. Cet ensemble de programme correspond typiquement à une bibliothèque de procédures développées dans le cadre d'un domaine particulier (e.g. le traitement d'images satellites, l'imagerie médicale).
Le système de pilotage de programmes va collecter dans une base de connaissances l'expertise relative aux possibilités de traitements offertes par un ensemble de programmes pour répondre à une application donnée. Il va exploiter ces connaissances conformément à la démarche mise au point par un expert grâce à un moteur de raisonnement conçu pour choisir et exécuter les programmes permettant de résoudre un objectif particulier de traitements de données.

système réactif :[Harel 85]
un système informatique qui réagit continûment avec son environnement, à la vitesse déterminée par cet environnement.

système temps réel : un système en interaction avec son environnement. Les interactions entre un système temps réel et son environnement peuvent s'effectuer soit à des moments déterminés par un référence de temps interne au système (système piloté par le temps ou *time-driven system*), soit à des moments déterminés par l'environnement lui-même (système piloté par les événements ou *event-driven system*.
Pour un système temps-réel, l'exactitude des applications ne dépend pas seulement de l'exactitude du résultat mais aussi du temps auquel ce résultat est produit. Il est donc essentiel de pouvoir garantir le respect des contraintes temporelle du système.

système de vision : système qui fonctionne de manière continue avec son environnement et qui communique avec lui à travers des capteurs visuels et/ou une interface utilisateur. Un tel système doit exécuter des actions de perception dédiées à l'application pour laquelle il a été conçu. De plus, ce système doit délivrer les résultats dans un délai fixe.

tâche de vision : élément de commande d'exécution d'unités d'organisation de programme.

temps réel : on dit qu'une application est soumise à des contraintes temps réel si ses résultats respectent certaines contraintes temporelles, i.e. les résultats sont fournies suivants des échéances données.
On distingue le temps réel *dur*, pour lequel aucun dépassement d'une échéance n'est toléré ; et le temps réel *lâche* dans le cas où des dépassements occasionnels des échéances ne met pas le système en difficulté.

thread : (ou processus léger)
Un thread est une entité d'exécution interne à un processus et qui possède un environnement simplifié. Les threads n'existent qu'à l'intérieur d'un processus. Ainsi si le processus meurt, les threads qu'il contient disparaissent en même temps que lui. Tous les threads d'un processus partagent l'ensemble des ressources de ce processus, permettant une programmation simple et efficace.

validation : la validation d'un système consiste à vérifier qu'il correspond bien à ses spécifications.

validité : [Meyer 88]
La validité est l'aptitude d'un produit logiciel à réaliser exactement les tâches définies par sa spécification.
La validité est une qualité essentielle mais difficile à atteindre car les spécifications sont difficiles à formaliser rigoureusement.

vérifiabilité : [Meyer 88]
La vérifiabilité est la facilité de préparation des procédures de recette et de certification (e.g. les tests et les procédures de debogages)