DEA ARAVIS 1996 : Stage de Fin d'Année

INRIA Sophia-Antipolis - Projet ROBOTVIS

2004, Route des Lucioles

BP 93 - 06902 Sophia Antipolis Cedex

France

ETUDE et DEVELOPPEMENT d'une MAQUETTE de GENERATEUR de CODE en VISION ROBOTIQUE

Soraya ARIAS

Responsables

DEA ARAVIS : Michel Barlaud

Stage : Eve Coste-Manière

Thierry Despeyroux

Thierry Viéville

Remerciements

Je tiens à remercier

Eve Coste-Manière, Thierry Despeyroux et Thierry Viéville d'avoir accepté de co-encadrer ce stage quelque peu original sinon par le fond, en tout cas par la forme, ainsi que de l'intérêt constant qu'ils ont manifesté pour ce travail.

Konstantin Kapellos et Nicolas Turro pour leur constante disponibilité et les discussions fructueuses que nous avons eues sur le langage ESTEREL et le système ORCCAD.

Olivier Faugeras et le projet Robotvis pour leur accueil au sein de leur équipe.

Reyes Enciso et Angeles Lopez grâce auxquelles mon stage a revêtu des couleurs espagnoles.

les stagiaires ARAVIS pour leur sempiternelle bonne humeur et pour avoir réussi à supporter mon caractère parfois difficile.

Résumé

Afin d'assurer l'exploitation optimale et efficace des différentes méthodes d'analyse d'image et de vision par ordinateur rencontrées dans de nombreuses classes d'applications, ce mémoire de DEA propose un ensemble de solutions théoriques et pratiques pour permettre leur spécification et leur implémentation. Elles sont basées sur :

- une réflexion internationale à travers le système IUE, sur la standardisation des méthodes rencontrées dans le monde de la vision par ordinateur ;
- une méthodologie de spécification qui emprunte des *concepts orientée-objet* pour la structuration des données et des méthodes associées, et des *méthodes formelles de spécification et de validation des systèmes réactifs* basées sur la programmtion synchrone ;
- une proposition de génération automatique de code *temps réel* respectant la norme POSIX

Ces solutions sont illustrées et validées en élaborant un prototype caractéristique pour la conception d'environnement de développement d'applications de vision temps-réel. La confrontation des aspects théoriques avec les exigences d'une mise en oeuvre pratique ouvre de nombreuses perspectives à ce travail, énoncées en conclusion de ce rapport.

Table des Matières

Liste des Figures

Chapitre 1

Introduction

Au cours des dix dernières années, la robotique, l'analyse d'images et les systèmes de vision par ordinateur sont devenus des éléments courants utilisés dans des domaines industriels très variés tels que l'évaluation et le contrôle de qualité, l'inspection autonome utilisant des robots, la reconnaissance et la classifications d'objets manufacturés, l'imagerie microscopique et médicale, ou encore l'imagerie satellite. Très vite, il s'est avéré impératif de pouvoir assurer une meilleure exploitation des méthodes développées par les équipes de recherche et les sites industriels qui travaillent dans des environnements temps-réel, et qui utilisent ces modules. Sans de tels outils, de nombreux développements restent lettre morte, soit parce qu'ils ne survivent pas au départ de leur concepteur, soit parce qu'ils restent des produits "académiques", inutilisables dans l'industrie.

Pour éviter ces problèmes, à l'échelle mondiale, des efforts ont été entrepris et continuent à l'être pour standardiser les structures de données, que ce soient pour les échanges en-ligne ou hors-ligne, ou pour les méthodes traitant de ces données. Un exemple très significatif de cet effort est la création de *IUE* ou *Image Understanding Environnement* [13] qui a été développé, sous l'impulsion du *ARPA*, dans le but de fournir une infrastructure logicielle orientée-objet d'outils de développement d'interface utilisateur et d'algorithmes de traitement et analyse d'image nécessaires pour mettre en pratique les résultats de la recherche dans le domaine de l'analyse et le traitement d'image. Aussi, IUE est-il en train de s'imposer dans le domaine de l'analyse d'images, que ce soit en symbiose avec d'autres initiatives telles que *Target Junior*, ou au-dessus d'outils de plus bas niveau tels que *Khoros*, ou encore de librairies telles que *KBV*. Mais jusqu'ici, aucun de ces efforts n'a encore abouti à créer, pour des systèmes temps-réel, un standard valable aussi bien pour le milieu de la recherche que pour l'industrie. C'est dans cette optique que se situe ce travail : **donner les caractéristiques d'une standardisation des structures de données et des méthodes qui les manipulent pour des systèmes soumis à des contraintes temps-réel**.

Nous travaillerons ici à partir d'un exemple précis, caractéristique des applications de la vision robotique : un système de vision active [5], muni d'une caméra contrôlable interfacée avec un environnement robotique élémentaire, qui doit enchaîner plusieurs tâches telles que détecter un objet en mouvement, suivre un ou plusieurs objets, passer d'un objet à un autre, déterminer des caractérisiques géométriques et cinématiques des objets observés [1].

D'un point de vue méthodologique, nous pensons qu'il est indispensable de travailler, *d'abord* avec un système concret et réel, typique d'applications du domaine tel que -pour ce qui nous concerne- la surveillance et la perception visuelle d'un environnement dynamique. A partir de cette étude expérimentale nous espérons pouvoir dégager des concepts généraux valables pour des applications connexes.

Cet exemple, dont on a conçu un prototype, nous a amené à dégager des éléments caractéristiques, donc espérons généraux, qui répondent aux besoins exprimés au niveau de la standardisation des données et des méthodes pour des systèmes tels que IUE, mais gérant le temps-réel.

Ce sont d'une part la *méthodologie objet* (et pour ce faire nous proposons un formalisme sous forme d'une stratification des spécification objets usuelles), et d'autre part *la réactivité* pour gérer le comportement du système temps-réel (réactivité qu'il sera intéressant de gérer à travers une méthodologie particuliere impliquant l'utilisation du langage ESTEREL [7]qui répondra au problème posé). Enfin, la portabilité du système, assurée par le biais de la norme POSIX [16], sera considérée.

Ces différents points seront analysés dans de ce document, à partir du plan suivant :

- dans une première partie, nous exposerons et analyserons plus en détails le contexte dans lequel se situe ce travail ;
- dans la deuxième partie, nous exposerons les concepts, liés au formalisme objet, que nous avons développés ;
- la troisième partie sera consacrée à la réactivité et à l'intégration de méthodes formelles de ce domaine au cadre de la perception visuelle ;

– enfin, nous présenterons en détail le prototype qui a été créé et les conclusions que l'on peut en tirer pour revenir à un cadre théorique général donnant une solution au problème posé

Nous conclurons sur ce travail en ouvrant des perspectives de recherche à moyen terme sur ce sujet.

Chapitre 2

Contexte de cette Etude

Comme on l'a mentionné dans l'introduction de ce document et comme il a été reconnu par la communauté scientifique concerné par le sujet réuni lors d'une session de travail qui est à l'origine de ce sujet et que nous réferrons sous le nom de projet *ReViMoGe*[1], le besoin de *standardisation* des méthodes et des structures de données est comblé au niveau des systèmes, qui ne sont pas soumis à des contraintes temps-réel, à travers le standard IUE. Ce n'est néanmoins pas le cas pour les systèmes temps-réel et on appellera *ReViMoGe* le projet d'un tel système, ébauché par ce groupe européen et dont ce travail de stage contribue donc à faire l'analyse.

Plus précisément, IUE ou *Image Understanding Environment* est un environnement logiciel de programmation pour la recherche en traitement d'image et vision artificielle. Ce projet est issu, à l'origine, d'un projet sur cinq années, financé par le département de recherche de la Défense américaine, le ARPA.

Le but premier d'IUE est d'accroître l'efficacité de la programmation (définition d'objets et de méthodes couramment employés) et l'échange de résultats de recherche dans ce domaine (données, algorithmes, etc $\cdots$). A cette fin, IUE fournit :

- une hiérarchie extensive de classes C++ représentant une large gamme d'objets, images, caractéristiques d'images et algorithmes, utilisés en vision et traitement

[1]Ce chapitre est la synthèse et une analyse personnelle d'une réunion de travail sur "European Contributions to IUE, for a preliminar analysis of the Real-Time Vision Module Generator project", menée à l'INRIA, en janvier 1995, avec la participation de *Mike Brady, Henrik Christensen, Fabrice Clara, Jim Crowley, Jan-Olof Ekhlund, Olivier Faugeras, Goesta Grandlund, Franck Mangin, Hans-Helmut Nagel, Marc Proesmans, Jan Nielsen, Giulio Sandini, Thomas Skordas, Marc Van-Diest, Luc Van-Gool, Thierry Viéville, Harald Winroth.* Le résultat de leurs réflexions (non publié) est disponible sous forme d'un compte-rendu de réunion.

d'images,

- une interface standard modulaire orientée objet bien documentée, et basée sur des outils industriels existants,
- une implémentation d'algorithmes standards (détection de points, ...), actuellement en cours,
- la possibilité d'utiliser et de coordonner la puissance de systèmes déjà existant dans ce domaine.

Néanmoins, les spécifications introduites dans IUE *ne couvrent pas les problèmes inhérents au temps réel et à la réactivité inhérente aux systèmes qu'ils mettent en oeuvre.* En effet, il existe des besoins importants de standardisation en :

   – Représentation des données et protocoles de données pour les échanges en-ligne d'informations ;
   – Architectures permettant de construire des systèmes temps-réel ;
   – Transfert simplifié depuis une plateforme expérimentale vers des installations industrielles temps-réel.

En outre, le développement d'un tel environnement, dans le contexte temps-réel, demande de pouvoir résoudre deux types de contraintes :

- Au niveau de la **Spécification** :
  Le problème semble être de standardiser différents types de données et d'informations de contrôle de façon à obtenir dans un même temps des représentations flexibles et communes des données. Ces représentations ne doivent pas seulement servir à décrire les données mais aussi les tâches, les flots de contrôle et les interactions fortes entre le logiciel et le hardware.
  Les définitions de telles spécifications vont permettre :
   – la modularité au niveau des méthodes orientées-objet ;
   – l'instantiation "rapide" des opérateurs génériques ;
   – la gestion d'événements et sa mise au point sur des systèmes cibles temps-réel.

- Au niveau de l'**Implémentation** :
  Le problème se situe dans la manière d'appliquer dans le monde réel de telles spécifications. Il faut pouvoir assurer que :
  - le système supporte le temps-réel afin d'être utilisé aussi bien pour les applications industrielles que pour celles de la recherche ;
  - le système soit suffisamment flexible pour pouvoir être utilisé dans différents types d'applications ;
  - le système soit séduisant vis-à-vis des programmeurs, c'est-à-dire que l'apprentissage du système soit rapide et simple ;
  - le système doit pouvoir fournir des méthodes simples pour la mise-à-jour et permettre d'inclure ou tester facilement de nouvelles méthodes.

Pour atteindre les objectifs fixés précédemment, voici les points clés qui seront à la base de notre travail :

- **La Modularité** : il faut un environnement modulaire pour garantir de pouvoir développer un système dont la taille ne sera fonction que de l'application envisagée et non de tout l'environnement. Précisons cette idée :
  - De la modularité découle la capacité d'utiliser conjointement ce qui a été précédemment développé et ce qui le sera au fur et à mesure des développements.
  - La modularité est importante à la fois dans le développement d'applications et pour améliorer les spécifications.
  - Mais il est aussi important, pour créer des applications embarquées, de ne pas avoir à "charger" plus que ce qui est déjà utilisé, pour avoir une taille minimale des codes.
  - De plus, cela permettrait un "apprentissage progressif" de telle sorte que le premier venu n'aie pas besoin d'apprendre plus qu'il n'est nécessaire pour débuter son application.
- **Les Restrictions Temps-réel** : l'utilisation du temps-réel impose au système de lourdes restrictions au niveau de la conception et des spécifications. Néanmoins, loin d'être négatif cet état de fait a l'énorme avantage suivant : en forçant à être le plus "simple possible", il permet de limiter les développements particuliers ou trop spécifiques donc de rester "générique". Ainsi :

- Même si chaque application a ses exigences propres en matière de temps-réel, et nécessite peut-être des caractéristiques particulières, la plupart de ces exigences ont une base commune. Par conséquent, on peut supposer pouvoir spécifier *une* architecture générale qui va servir à construire des systèmes de contrôle temps-réel pour différentes applications.
- Ceci va permettre aussi de simplifier la construction spécifique d'un système de contrôle. L'architecture ne devra pas seulement gérer un ensemble de ressources (telles que les processeurs, la mémoire, les canaux de communications ...etc) utilisées par chaque partie de l'application, mais il devra aussi gérer le temps d'utilisation de ces ressources et leur séquencement.

En fait, le développement d'une application nécessite la *définition formelle* d'un ensemble de *méthodes* qui permettent la mise en oeuvre d'un ensemble de *concepts* théoriques, sur lesquels repose la spécification des entités que l'on souhaite manipuler (ex : algorithme,
etc...).

L'implémentation de ces méthodes reposent quant à elles sur le choix d'un ensemble *d'outils* adaptés et efficaces.

Le travail consiste à fournir ces outils dans un environnement intégré selon une architecture bien définie, pour permettre à différentes personnes d'interagir et de modifier les entités manipulées indépendament, tout en cachant les détails d'implémentation requis pour suivre les méthodes proposées et les concepts sous-jacents.

Un autre facteur important, pour atteindre les objectifs dégagés par les chercheurs, lors de la réunion de travail de janvier 1995, est d'aboutir à *un coût de développement d'un système donné le plus bas possible.* Un moyen pour y parvenir consiste à utiliser des outils déjà existants. Ainsi, en plus de réduire le coût de développement, on minimisera le degré d'incertitude sur le système et assurera sa pérénité au fur et à mesure de l'évolution des matériels et logiciels.

Cependant, il est bon de noter que cette problématique implique d'autres développements encore du domaine de la recherche tels que :

- La génération automatique de code (à partir de spécifications générales des données ou des flots de contrôle).

- La modélisation orientée-objet de données, de flots de contrôle, de ressources logiciels etc...
- Les techniques de programmation synchrone pour spécifier la réactivité, les opérateurs de parallélisme, l'interface avec les événements externes, etc ...
- Les opérateurs génériques pour le parallélisme pour pouvoir installer les modules développés sur des machines parallèles MIMD.

Ces aspects sont fortement liés avec ce type de projet et pourraient être utilisés pour optimiser le système. Ils feront l'objet de travaux ultérieurs.

Considérant les différents critères mis en avant par les partenaires consultés, les outils actuellement disponibles et utilisés sur lesquels on peut s'appuyer, pour l'élaboration du système envisagé, sont les suivants:

- **Maple** comme générateur de code :
  Étant donné que l'on va faire de la génération de code de type calculatoire, on a besoin d'effectuer beaucoup de calculs symboliques tel que : résoudre des équations, dériver de manière symbolique, réduire des expressions, propager des constantes à travers le code, créer des opérateurs génériques, etc... [3, 4]. Et bien entendu, le meilleur outil pour réaliser ces manipulations est un calculateur symbolique. C'est pour cela que Maple5.3 va être employé pour cette tâche.
  Étant donné les caractéristiques de base de ce système symbolique, le travail additionnel à effectuer sera simplement de :
  - ajouter une couche dite "orientée-objet" au-dessus du calculateur symbolique
  - concevoir des "mécanismes de génération de code" utilisant les définitions orientées-objet établies ci-avant,

  ce qui est en cours à travers l'extension `Mascotte/mso` de Maple [2].

  Deux autres raisons pratiques favorisent l'utilisation de Maple :

  1. c'est un puissant interpréteur doté d'un grand nombre de primitives permettant la manipulation de données symboliques, non nécessairement mathématiques ;
  2. c'est un produit facile à apprendre, bon-marché, bien reconnu et populaire, tout en étant notoirement efficace.

- Un langage **C-restreint** comme langage cible :
  On a choisi de générer le code en utilisant un *sous-ensemble du langage C standard* que l'on a baptisé *C-restreint* ou encore ρ-*C*.
  L'utilisation d'un sous-langage C est motivé par les faits suivants :

  – chaque processeur sur le marché possède un compilateur C. Il est le plus souvent adapté à l'architecture du processeur de façon à ce que l'usage des registres CPU, de l'adressage mémoire et des instructions soient optimisés.

  – le but recherché avec le système *ReViMoGe* étant de créer un standard "universel", le sous-langage C permettra de rendre le système portable sur presque n'importe quel type de machine ou de processeur.

  – d'un point de vue plus théorique, on définit ainsi une "machine virtuelle" plus que réaliste puisque instantiable sur toutes les architectures, mais sans avoir à modéliser tout ce qui concerne l'architecture d'un processeur donné (micro-instructions, mémoire cache, etc...) qui serait d'une complexité rédibitoire, à ce niveau.

  Mais pourquoi utiliser un *sous-ensemble* de C ? Il y a deux principales raisons à cela:

  – on n'a pas besoin dans la génération de code de l'intégralité du langage C.

  – on évite comme cela d'utiliser certaines caractéristiques du langages qui sont au niveau sémantique très lourdes et mal définies (pointeurs, casting, etc...).

  Nous développerons plus en détails ce sous-langage à l'annexe B.

- **Esterel** pour définir la partie réactivité du code :
  Pour la partie réactive du système, le langage C n'est pas suffisant à lui seul pour définir nos outils, car il n'a pas été conçu pour pouvoir gérer du synchronisme ou encore du parallélisme. Aussi a-t-on choisi d'utiliser les primitives du langage Esterel, pour les motivations suivantes :

  – Esterel permet une gestion optimale des événements, et par conséquent garantit une grande efficacité au niveau de l'exécution, car ce langage se base sur une hypothèse de synchronisation qui permet de produire un automate d'états finis pour le contrôle de la réactivité de toute application et donc de résoudre à la compilation les problèmes de communication et de séquencement de processus.

– Esterel permet de prouver, de part son élaboration même, de manière mathématique, que le programme aura bien le fonctionnement voulu, lors de l'exécution. Par conséquent, cela permet d'introduire des preuves au niveau du code.

- **Posix** pour le formalisme temps-réel :
  POSIX ou **P**ortable **O**perating **S**ystem **I**nterface est une norme qui fournit des moyens de rendre portables des applications écrites pour un système d'exploitation donné, et de rendre l'écriture d'un programme plus simple. Toutes les primitives temps-réel utilisées dans notre système sont compatibles avec la norme Posix. Par conséquent, elles peuvent tourner sous tout système compatible avec Posix tel que VxWorks et même avec certains packages logiciels sur PC.

- Les primitives **PVM** pour définir le parallélisme :
  Le langage PVM est devenu un outil relativement bien accepté pour définir des processus parallèles et les communications entre ces processus. C'est ce langage qui a été retenu pour définir les modules de vision que l'on va lancer sur différents processeurs. En fait, grâce au formalisme de programmation parallèle basée sur le langage **Esterel**, seul un petit sous-ensemble de primitives PVM sera utilisé.

- **Java** pour définir l'interface utilisateur :
  L'interface utilisateur sera réalisée grâce au langage Java, et à ses "widgets". Cette méthode permet à l'utilisateur de spécifier assez simplement les menus, les pannels interagissant avec l'utilisateur, les données et images de sortie. Il suffit à l'utilisateur de créer son interface "à la main" et Java se charge de l'implémenter. De plus, on peut obtenir une interface efficace sur une station de travail et proposer des démonstrations du système sur le réseau.

- **WWW** pour la documentation et la distribution du produit :
  La documentation des modules sera générée en "hypertext" en utilisant le langage HTML.

  Le choix de WWW permet d'éliminer une partie des problèmes liés à l'interface utilisateur. De plus, la diffusion, la publicité et la maintenance du logiciel sont facilement réalisées à travers INTERNET étant donné que les données WWW et sa hiérarchie peuvent être directement transférées à travers le réseau ou vers n'importe quel type de support, y compris les modules Java. D'autre part, l'échange de données et de méthodes entre utilisateurs est facile à établir.

Conclusion

En conclusion à cet état des lieux en matière d'environnement logiciel de programmation, il semble que, pour élaborer un système de la même envergure que IUE mais qui couvrent les aspects temps réel, on dispose déjà de beaucoup d'outils (ESTEREL, MAPLE, POSIX, etc $\cdots$) susceptibles d'intervenir dans l'élaboration d'un tel système.

Cependant, il manque encore une méthodologie de spécification compatible avec toutes les exigences répertoriées dans ce chapitre, d'une part, des spécifications modulaires, réactives et hiérarchiques, et, d'autre part, de gérer de manière homogène et fiable tous les outils répertoriés.

Par conséquent, la définition de spécification combinant concepts objet, réactivité et portabilité temps réel font l'objet principal de ce travail de DEA.

Chapitre 3

Stratification de spécifications objet

3.1 Définitions et Intérêt de l'approche Objet

Comme avec toute spécification orientée objet le but recherché est :

- obtenir un modèle qui permette la coopération synchrone et adaptative d'objets utilisés en vision.
- élaborer des systèmes informatiques dont la complexité s'avère être beaucoup plus complexe que ce que peut gérer seul un cerveau humain.

Dans ce but, il faut pouvoir concevoir un outil qui permette de spécifier une hiérarchie de composants logiciels parfaitement modulaires, cohérents entre eux, persistants et couplés entre eux de manière flexible et minimale. Or, l'approche objet permet l'amélioration de l'architecture des programmes car elle permet :

- une décomposition modulaire rigoureuse reposant sur des entités appelés *objets* manipulées par l'application. (*modularité*)
- la séparation entre l'interface abstraite d'utilisation de chacun de ces objets et les détails de leur implémentation. (*encapsulation*)

Par conséquent, pour la conception de notre système, nous allons utiliser les concepts de base de la programmation orientée-objet que sont :

- **L'abstraction** : dans notre spécification, l'abstraction existe à travers l'utilisation d'objets que l'on nomme des *minimal symbolic object* ou *mso* qui vont permettre de définir des classes d'objets. Un *mso* est en fait une liste d'équations $\{attribut = valeur\}$. Les méthodes associées aux objets seront manipulées à travers les attributs spécifiques à chaque *mso*.

- **L'héritage et le polymorphisme** : ces propriétés seront réalisées à travers la notion de *parents* d'un objet. Les *parents* vont permettre de masquer une partie du contenu d'un attribut. De manière interne, *valeur* se récupère à travers des liens d'héritage, que l'on peut modifier et|ou compléter.
- **La modularité et la hiérarchie** : étant donné que chaque liste d'équations est indépendante, le couplage entre les différentes parties de la hiérarchie est uniquement fourni par la classe attribut ou la référence par valeur. C'est en tout point similaire aux liens sous le système WWW ou dans un système de fichiers.
  D'un point de vue hiérarchie, on peut définir un ordonnancement des différentes classes puisque qu'une classe peut-être sous-classe d'une autre. De plus, le mécanisme est relié d'une certaine manière au mécanisme de prototypage car une classe n'est, en fait, qu'une liste de valeurs par défaut ou de valeurs référencées par attributs.

Le type des objets est aussi une notion donné par les attributs de la classe. Le type d'un objet est sa propre classe quelque soit les attributs qui la composent. Les types paramétriques correspondent à une classe dont une valeur est une expression intégrant d'autres attributs.

Pour une définition plus détaillée des concepts orientés-objet introduits ici, vous pouvez vous référez à l'annexe A ou encore pour plus d'information à [17] et [18].

3.2 *MSO* : Base de la Structuration Objet Stratifiée

Nous allons, dans ce paragraphe, exposer les concepts élaborés qui vont permettre de structurer le système.

3.2.1 Concepts et Définitions :

Définition I : *Minimal Symbolic Object*

Un *minimal symbolic object* ou *mso* est une liste d'*attributs* qui ont chacun une valeur donnée. Chaque valeur est constituée par une liste d'équations appelées des *facettes*.

Exemple:

```
Obj  =  { Attribute1 = Value1, Attribute2 = Value2, ···}
            <-     Facet 1     -> <-     Facet 2     ->
```

Définition II : *Valeur d'un Attribut*

La valeur d'un attribut peut être soit :

- la valeur par défaut :
  Si on demande l'évaluation d'un attribut qui est indéfini ou absent dans la définition du *mso*, soit :
  - la valeur de l'attribut est son propre nom. On utilise, alors, la même convention que celle du langage Maple.
  - la méta-valeur "NULL".
- une chaîne de caractères :
  Cette chaîne est composée de tous caractères imprimables y compris les chiffres.
  Par conséquent, une **propriété importante** est que les nombres sont représentés par des chaînes de caractères et peuvent avoir une précision infinie (dans la limite de la représentation de la mémoire de la machine employée).
- un (sous)-objet ou encore un (sous)-*mso*.

Par convention, *les fonctions, les listes et les ensembles* ont une représentation canonique en tant que *mso* :

	Représentation standard	Représentation mso
`Fonction`	$f(a_1, a_2, \cdots)$	$\{0 = f, 1 = a_1, 2 = a_2, \cdots\}$
`Liste`	$[v_1, v_2, \cdots]$	$\{1 = v_1, 2 = v_2, \cdots\}$
`Ensemble`	$\{v_1, v_2, \cdots\}$	$\{v_1 = 1, v_2 = 1, \cdots\}$

Pour un ensemble fixé d'attributs et de taille maximal connu, on peut affirmer que les *mso* sont isomorphiques à une structure C sans pointeur. Ceci assure, par conséquent, un accès constant dans le temps et pas d'erreur de dépassement mémoire.

Il ressort de ces définitions qu'elles recouvrent un très grand nombre de données informatiques et qu'une grande partie des informations logicielles peut être représentée par cette syntaxe simple.

On peut citer pour exemple :

– Les fichiers de données tels que les répertoires d'adresse ou les références bibliographiques, ou tout ensemble de donnée de type "fiche" ou "formulaire" :

```
{
 {
  Name = 'John Smith',
  Address = 'Po.Box. 999, New-Evil, 3-1416 Mass U.S.A',
  Tel = '+1 123 12 23 23'
 },
 {
  Name = 'Pierre Martin',
  Tel = '+33 45 67 89 01'
 }
}
```

– Les configurations de programmes et de ressources. Beaucoup de systèmes interactifs sont maintenant basés sur la définition d'une structure de données à partir de laquelle une interface ou une configuration est générée. C'est en particulier le cas des ressources X-server ou des définitions de l'interface Tcl/Tk :

```
{
 MenuName = 'General-Menu',
 MenuItems = {
  'Open File' = 'fileOpen(fileName, fileData)',
  .../...
  'Quit' = 'exit()'
 },
 .../...
}
```

– Les objets graphiques ou les primitives manipulées en vision peuvent aussi être définis de cette manière, comme le montre l'exemple ci-dessous, où l'objet `m12` est référencé dans la liste `Parts` de l'objet de type `'Graphic Object'` :

```
{
 Type = '2D-point'
 Name = m12
 x = 123, y = 125
}
{
 Type = 'Graphic Object',
 Parts = { '../m12', .../... },
}
```

Si l'on se place du point de vue du *style de programmation* induit par ces définitions, et en considérant un logiciel comme constitué de données et de structures de contrôle, il en ressort donc que :

- au niveau des données on va placer le plus d'information et de connaissance possibles sur le logiciel ;
- au niveau contrôle, on va minimiser la quantité d'information à traiter.

L'idée de base est, par conséquent, de spécifier le plus possible dans les *minimal symbolic objects* toute la connaissance que l'on souhaite voir apparaître dans le logiciel.

De plus, on pourra spécifier directement avec la syntaxe *mso* reliée à des outils standard les entités nécessaire à la vision, comme la documentation ou les interfaces mais ceci sans plus avoir à les programmer.

3.3 Du Statique vers le Dynamique

Dans le paragraphe 3.2, on a montré que les spécifications se basaient sur les *mso*, i.e des objets définis comme des listes d'équations $attribut = valeur$. Mais nous n'avons pas encore introduit de dynamisme pour permettre de faire évoluer ces objets.

Et c'est ce que l'on va expliciter dans cette partie.

3.3.1 Les Attributs Dynamiques :

Définition III :*Attributs Statiques et Dynamiques*

On considère maintenant que les valeurs des attributs peuvent être :

- soit statiques - voir la définition II ;
- soit dynamiques, i.e. les valeurs des attributs sont obtenues à la suite de l'évaluation d'une fonction que l'on appelle *straight-line function.*

Définition III bis : *Straight-line Functions*

Une *straight-line function* est un *mso* qui est défini récursivement par :

- une Référence Attribut : `obj.attribute` ;
- un appel de straight-line function : `f(args(s))` ;
- les opérateurs fonctionnels suivants :
  - l'opérateur de Condition :

    $$\begin{array}{|ll} if\ cond & then\ trueValue\ [elif\ otherCond\ then\ otherValue] \cdots \\ & else\ falseValue\ fi \end{array}$$

    `If(cond,trueValue,[otherCond,otherValue,]···falseValue)`

    qui correspond à l'opérateur C :

    `cond ? trueValue : falseValue`.
  - l'opérateur d'Itération :

    $$[Expr\ \$\ Index = firstValue..lastValue]$$

    `Do(Expr,Index,firstValue,lastValue)`

    qui permet de créer une liste d'expressions en substituant dans l'ensemble `Index` par `Expr` les valeurs `firstValue,···,lastValue`
  - l'opérateur Séquentiel :

    $$[Expr_1, Expr_2, ..., Expr_n]$$

    `Seq(Expr_1, Expr_2, ..., Expr_n)`

    L'expression de la séquence est évaluée de la gauche vers la droite et le résultat est la dernière expression.
  - l'opérateur Map :

    $$Operator\ :>\ List$$

    `Map(Operator, List)`

    La fonction *Map* applique l'opérateur à chaque élément de la liste.
  - les opérateurs usuels :

    `‘+‘, ‘*‘, ‘-‘, ‘/‘, ‘%‘, ‘&‘, ‘|‘, ‘¡, ‘=‘, ‘<>‘, ‘<‘, ‘>‘, ‘<=‘, ...etc`

Les *straight-line functions* peuvent être définies de manière :

- non-récursive : dans ce cas, ces fonctions ont comme propriétés :
  - d'avoir un temps d'exécution prédictif constant. Ce temps d'exécution est une fonction déterministe de la taille des données en entrée.
  - de ne pas produire de boucle infinie ni d'exception, comme celles dûes à des erreurs de gestion de la mémoire.
  - de pouvoir être implémentées par un morceau de code dédié uniquement aux calculs, appelé aussi *straight-line program* ou **SLP**, le code étant écrit en utilisant un langage C-restreint.
- récursive : dans ce cas, ces fonctions ont comme propriétés :
  - de correspondre à des fonctions de complexité quelconque ;
  - de permettre un style de programmation "fonctionnelle" élégant.

De ces définitions, on peut faire les remarques suivantes :

- les objets symboliques ou *mso* n'ont pas besoin de définir de "méthodes" comme c'est le cas dans l'approche objet classique. Ce sont les attributs dynamiques qui feront office de méthodes à travers le mécanisme d'évaluation des valeurs des attributs.
- les attributs dynamiques sont stockés dans une facette `Eval`.
- tous les objets peuvent être référencés, par convention, par un attribut unique appelé `Name`. Cette étiquette est globale pour un environnement donné ce qui implique que l'on devra utiliser des règles de nommage automatiques.
- on peut aussi envisager, pour ces attributs dynamiques, une implémentation sous forme de *call-back* C.

3.3.2 Héritage :

Définition IV : *Attributs Dérivés*

Les attributs, qui ne seront pas définis, seront considérés comme ayant leur valeur qui dérive d'un ou d'autres objets. Ces objets seront appelés "parents".

Le premier attribut défini chez un parent sera pris comme valeur de l'attribut non défini dans l'objet fils, ce qui permet de modéliser le mécanisme d'héritage.

On peut définir plusieurs parents selon un ordre séquentiel : on permet ainsi un héritage multiple mais séquentiel où un objet hérite en fait d'un parent puis d'un autre, puis d'autre etc...

La liste des parents est stockée dans une facette appelée : `Parents`.

La fermeture transitive des parents peut être évaluée. Par conséquent, et contrairement à l'implémentation orienté-objet traditionnelle, on a une vue complète de l'héritage et on est à même de pouvoir extraire des objets de la hiérarchie inhérente à l'héritage.

Grâce à ce mécanisme d'attributs dérivés, on est capable de modéliser les différentes propriétés des langages orientés-objet typés comme :

- le polymorphisme.
- la généricité : puisque les *straight-line functions* sont des *mso* standard et peuvent admettre des facettes formelles additionnelles.
- l'abstraction : les attributs dérivés peuvent être utilisés pour manipuler des informations que l'on cherche à cacher comme par exemple, les options d'affichage, les caractéristiques de configuration.

Voici résumé sur le schéma ci-dessus l'imbrication des différents niveaux exposés dans ce chapitre :

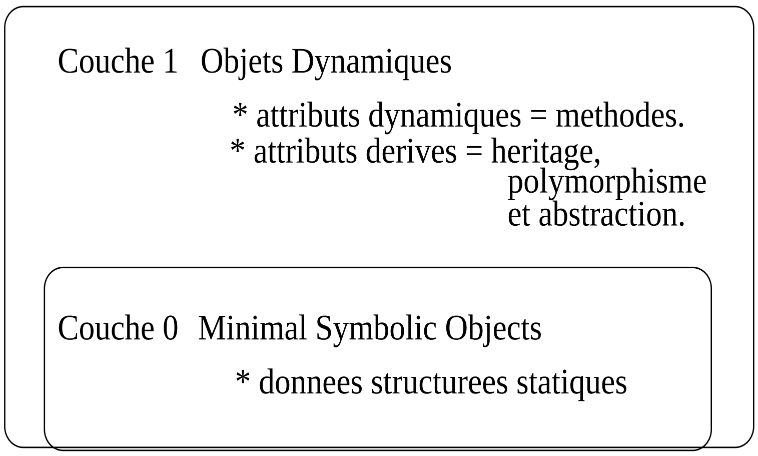

Figure 3.1: Stratification au niveau Spécification du système *ReViMoGe*

Chapitre 4

Méthodes Réactives Synchrones

Si on reprend l'exemple qui a été donné à l'introduction (au chapitre 1), du système de vision active, qui permet de détecter des objets en mouvement, de suivre un ou plusieurs objets, etc..., le problème qui se pose ici c'est de pouvoir :

- enchaîner les différents modules de vision et de robotiques, qui composent l'application, en fonction des requêtes générées par le système ;
- contrôler le fonctionnement de ces modules ;
- structurer leur fonctionnement en fonction de la tâche perceptuelle à réaliser ;
- assurer que le contrôle soit robuste et fiable et que les contraintes de temps soient bien respectées afin que l'enchaînement des modules respectent les contraintes temps-réel du système.

Pour ce faire, il est nécessaire d'introduire la notion de systèmes réactifs, et à l'intérieur de cette notion de réactivité, nous allons nous intéresser plus particulièrement aux approches synchrones, qui permettent de les modéliser, et à un des langages de cette classe, le langage ESTEREL.

4.1 Les Systèmes Réactifs :

Dans la notion de systèmes réactifs a été introduite par [19] et fait référence à des systèmes qui sont en perpétuelle interaction avec leur environnement et réagissent à la vitesse de ce même environnement. Pourquoi s'intéresse-t-on aux systèmes réactifs dans le cadre de notre travail ? La réponse est liée à la notion de comportement d'un système qui est utilisée dans le cadre des applications de vision qui nous intéressent, qui mélangent à la fois des aspects robotiques avec les aspects modules de vision. Le comportement dans un système réactif est entièrement spécifié par l'ensemble des séquences permises d'événements externes d'entrée et sortie qui constituent l'interface du système avec son environnement. Présentons d'abord les caractéristiques des systèmes réactifs.

4.1.1 Caractéristiques :

Le déterminisme. La spécification fonctionnelle des systèmes réactifs est déterministe, en ce sens que la même séquence d'entrées produit toujours la même séquence de sorties. Il n'y a pas de raison pour qu'un système comme ceux décrits plus haut ait un comportement imprévisible. Étant donné que les systèmes déterministes sont plus faciles à comprendre, analyser et diagnostiquer, le déterminisme doit être préservé le plus loin possible depuis la conception vers l'implémentation.

Les contraintes de temps. Les contraintes temporelles que doit respecter un système réactif proviennent du fait qu'il doit réagir au rythme de son environnement. Ceci demande la prise en compte de deux visions différentes du temps. Le temps comme *durée*, qui concerne à la fois le rythme auquel les entrées doivent être prises en compte et la vitesse de la production de sorties comme réaction aux entrées; le temps comme une suite d'*instants*, auxquels le système doit déclencher ou arrêter l'exécution d'une activité; cet instant peut être lié à une spécification non nécessairement temporelle (i.e arrêter avant 30 mètres). Ces contraintes doivent être exprimées à la spécification du système, prises en compte pendant sa conception et testées sur son implémentation. Elles doivent être impérativement satisfaites, ce qui demande une implémentation efficace et une évaluation précise du temps de réaction des programmes.

La fiabilité. Les contraintes de fiabilité proviennent de l'utilisation qui est faite des systèmes réactifs. Ils interviennent en effet dans le contrôle de dispositifs physiques qui peuvent être critiques (systèmes instables, centrales nucléaires, TGV, avions, ...). Il est donc important de pouvoir vérifier formellement les propriétés d'un système réactif afin d'éliminer ses sources d'erreurs potentielles. Des méthodes de conception et des outils qui supportent l'utilisation de méthodes formelles doivent être utilisées.

Le parallélisme. Le parallélisme est une caractéristique intrinsèque des systèmes réactifs puisque système et environnement évoluent en parallèle. De plus, la plupart des systèmes réactifs se décomposent naturellement en unités qui évoluent en parallèle et qui communiquent entre elles afin de produire un comportement donné. La prise en compte de la concurrence doit être faite de manière modulaire sans que le déterminisme du système soit remis en cause.

4.1.2 Méthodes de Programmation :

Au départ, la programmation des systèmes réactifs s'est basée sur des implémentation physique. Mais du fait de la complexité croissante des applications, les solutions logicielles se sont développées. En laissant de côté la programmation bas niveau (en langage assembleur), on peut distinguer deux types d'approches :

(i) l'approche basée sur un modèle formel : les systèmes de transitions. Cette approche a l'avantage d'être déterministe et d'avoir de bonnes performances à l'exécution. Cependant, elle ne permet pas d'exprimer explicitement le parallélisme : la gestion du parallélisme est très lourde et est source d'erreurs.
Exemple : les automates, les réseaux de Pétri.

(ii) l'approche basée sur des outils permettant d'exprimer la concurrence. Cette approche est asynchrone et offre des mécanismes de communication et de synchonisation entre processus parallèle. Mais elle a l'inconvénient de ne pas être déterministe et du fait de son asynchronisme la gestion du temps est souvent floue.
Exemple : les systèmes temps-réel, les langages de programmation concurrente tels que les langages parallèles ADA, OCCAM.

Par conséquent, et toujours selon [14], en regard à ses deux approches et à leurs principales propriétés, un utilisateur a le choix soit entre une approche déterministe, soit entre un approche asynchrone permettant le parallélisme.

Cependant, selon nous, il existe une autre approche qui réconcilie la programmation déterministe et la programmation concurrente. Cette approche permet la spécification, la conception et l'implémentation des systèmes réactifs et permet de programmer dans un langage de haut-niveau intégrant le parallélisme, tout en bénéficiant du déterminisme et de bonnes performances à l'exécution, cette approche est *l'approche synchrone.* C'est cette approche que nous allons aborder dans le paragraphe suivant 4.2, et nous focaliserons notre attention sur le langage ESTEREL qui nous paraît être le bon candidat pour utiliser cette approche synchrone dans notre contexte.

4.2 L'approche Synchrone et le Langage Esterel

4.2.1 L'approche Synchrone :

Ce sont Berry et Gonthier [6] qui ont développé cette approche synchrone. Reprenant [14] de nouveau, nous pouvons dire que cette approche s'appuie sur une hypothèse importante : *l'hypothèse de synchronisme.* Cette hypothèse consiste à considérer que chaque réaction d'un système réactif est instantanée. Le système produit ses sorties de manière synchrone aux entrées. C'est comme si l'on disposait d'une machine si rapide qu'elle pouvait exécuter les opérations liées au séquencement des instructions, à la gestion des processus, aux communications inter-processus et aux traitements de données élémentaires dans un temps nul (non observable).

Ainsi, le déroulement du temps n'est plus lié à l'exécution du programme mais uniquement à l'occurrence du flux d'entrée qu'il traite. Une occurrence définit un *instant* de l'exécution du système. Le "temps physique", qui joue un rôle si particulier dans les approches asynchrones, n'a plus de raison d'être privilégié: pour le manipuler il suffit d'écouter un signal d'entrée particulier produit par une horloge. Ceci permet une double homogénéisation: d'une part le temps se manipule comme un signal normal, d'autre part tout signal peut se manipuler comme définissant son propre temps, ce qui correspond à la notion du *temps multiforme.*

Dans l'approche synchrone, on parle d'*instant* dans le sens instant logique; l'histoire du système est une séquence totalement ordonnée d'instants logiques. A chacun des instants, on peut avoir l'occurrence de zéro, un ou plusieurs événements. Les événements présents au même instant logique sont considérés comme simultanés. A l'exception de ces instants rien ne passe ni dans le système ni dans l'environnement. On rejoint ainsi le modèle du comportement d'un système réactif.

D'un point de vue plus pratique, on peut dire que l'hypothèse synchrone

- permet de rendre les langages synchrones déterministes. Or les systèmes réactifs sont purement déterministes; les langages synchrones permettent de les programmer réellement ainsi, et donc de simplifier considérablement leur réalisation, leur mise au point et l'étude de leur comportement.

- permet de simplifier le raisonnement sur le temps et de le manipuler de manière naturelle et exacte. En effet, les différentes parties du programme communiquant par diffusion instantanée d'information, s'interrompent instantanément et provoquent des exceptions instantanément. La notion de temps multiforme introduit une souplesse de programmation et une possibilité d'abstraction sur les applications à développer. En outre, il existe de nombreuses primitives temporelles dans les langages synchrones qui permettent de manipuler les exceptions, les délais, ainsi que la suspension d'exécution.

- permet la validation des programmes engendrés. Ceci est dû à la sémantique formelle sur laquelle sont construits ces langages, mais cela tient aussi au déterminisme des codes produits par les divers compilateurs. La compilation d'un programme synchrone génère un automate à états fini séquentiel, déterministe et en général minimal. L'avantage est double. D'une part le code produit est efficace. Le programme parallèle initiale est traduit en un programme purement séquentiel équivalent. L'exécution d'un tel programme n'introduit pas de gestion de processus et les communications internes sont compilées et codées dans l'automate. Le temps maximal de la réaction est prévisible, et, ainsi, la validité de l'hypothèse synchrone peut être testée. D'autre part, une fois l'automate construit, il est possible d'utiliser des outils qui travaillent sur les automates pour pouvoir procéder à des vérifications formelles et s'assurer que le programme implémente bien les spécifications voulues.

Par conséquent, si on reprend notre exemple de système réactif donné dans l'introduction à ce chapitre, qui a à gérer l'interface avec l'environnement, à décider des actions à enchaîner et des calculs à entreprendre, il serait intéressant d'utiliser un langage synchrone pour s'occuper du noyau réactif du système, c'est-à-dire du séquencement et des synchronisations logiques du système. Or, il existe déjà dans le domaine robotique de nombreux exemples ([11], [9]) où de tels noyaux réactifs ont été employés, et ont montré que le formalisme synchrone permettait de modéliser aussi les phénomènes présentant de l'asynchronisme. Ces travaux ont été effectués autour du langage synchrone ESTEREL. Par conséquent, ESTEREL nous paraît un bon candidat pour spécifier et implémenter le type de système réactif dont nous avons besoin pour nos applications.

4.2.2 Le Langage Synchrone ESTEREL :

ESTEREL ([7], [21]) est un langage de programmation synchrone de haut niveau, textuel, de style impératif destiné à la programmation des systèmes réactifs.

Caractéristiques

- Un programme ESTEREL communique avec son environnement par l'intermédiaire de *signaux* et de *capteurs*. Les signaux sont utilisés à la fois comme entrées et comme sorties, tandis que les capteurs sont utilisés que comme entrées. A un signal sont associées deux informations: la *valeur*, qui est rémanente et les *tops* qui sont transitoires et indiquent les instants auxquels le signal est émis. Il existe des signaux dits "purs" qui n'ont pas de valeur et peuvent donc uniquement être émis ou non. Lorsqu'un signal est valué, sa valeur ne change que lorsqu'un top est émis. Ce top est utilisé pour propager et détecter la nouvelle valeur. Un capteur a une valeur mais pas un top.

- Un programme ESTEREL peut être composé de sous-modules qui communiquent par des signaux internes ou locaux (signaux qui ne sont pas perçus de l'extérieur). Les signaux d'entrée, sortie et locaux se manipulent tous de la même manière. Ils sont diffusés instantanément à toutes les composantes du programme.

- ESTEREL considère les systèmes réactifs avant tout comme des systèmes "à interruptions" dans lesquels l'extérieur commande le programme. La seule cause de réaction d'un programme est ainsi l'occurrence de signaux d'entrée. Les capteurs ne peuvent être lus, les signaux de sortie et locaux ne peuvent être émis que de façon synchrone avec la réception de signaux d'entrée.

- La notion de simultanéité est captée à travers le concept d'événement. Un événement est un ensemble d'occurrences simultanées de signaux.

Propriétés

- Au niveau de la programmation : ESTEREL introduit un style de programmation spécifique ([7]). Tout d'abord les programmes écrits en ESTEREL sont très proches de la spécification des problèmes qu'ils traitent. Ceci provient de l'approche multiforme du temps, adoptée par la programmation synchrone, qui en ESTEREL consiste en l'utilisation de tous les signaux comme définissant leur propre "unité de temps" pour compter des "délais".
  Ensuite, ESTEREL permet une programmation modulaire du fait de la diffusion synchrone des signaux. En effet, un module ESTEREL qui émet un signal n'a pas à savoir qui est en attente de ce signal et inversement. La modularité au niveau programmation permet d'implémenter des parties du comportement du système dans des modules différents. Ceci facilite la vérification globale du programme : de manière séparer on vérifie l'adéquation entre le comportement du module et sa spécification, puis on démontre que la coopération de ces modules implémente bien le comportement global spécifié.

- Au niveau de la compilation : le compilateur permet de traduire un programme ESTEREL en un automate fini. Cette traduction en un système de transitions étiquetées est élaborée à partir de la sémantique mathématique d'ESTEREL.
  L'automate produit par le compilateur ESTEREL est déterministe et en général minimal; le parallélisme et les communications qui apparaissent dans le code source sont compilées pour produire un code séquentiel. Les échanges de signaux internes n'apparaissent plus. Les avantages de la compilation en automate sont l'efficacité d'exécution et la prévisibilité du temps maximal de l'exécution d'une transition. Cette traduction produit une procédure dont l'appel provoque une réaction de l'automate. Pour activer cette procédure on doit écrire un programme principal qui procède de la façon suivante:

  1. il construit l'événement courant,
  2. il active l'automate qui exécute la transition correspondante,
  3. il retourne le contrôle au programme appelant.

- Au niveau de la gestion des tâches asynchrones : le besoin de la gestion par un programme synchrone de tâches qui prennent un temps physique non négligeable, c'est à dire de tâches qui mettent plus de temps à s'exécuter que le temps minimal entre deux occurrences d'événements, a amené l'introduction dans le langage de la primitive *exec*. Cette primitive permet de lancer une tâche externe, et d'attendre sa terminaison mais aussi elle permet d'envoyer une demande de terminaison à la tâche asynchrone lors que parvient une demande de terminaison forcée.

- Les *outils de vérification et de validation* : ils considèrent la structure interne de l'automate compilé. Ils permettent des comparaisons avec des spécifications ainsi que des visualisations globales ou partielles des états et des transitions. On utilise principalement AUTO et son interface graphique AUTOGRAPH.

Méthodologie

Cette possibilité de vérifier et de valider toute structure ESTEREL nous semble très importante pour le type de système que l'on cherche à créer dans le domaine de la vision. Car il est important de pouvoir prouver que le système aura bien le comportement spécifié au départ. Aussi serait-il intéressant d'utiliser une méthodologie à travers laquelle on puisse spécifier, implémenter et aussi valider un système réactif synchrone en utilisant le langage ESTEREL.

Un exemple d'une telle méthodologie est proposé à travers le système ORCCAD ou **O**pen **R**obot **C**ontroller **C**omputer **A**ided **D**esign, qui a été developpé par l'équipe *ICARE/INRIA* pour des environnements robotiques.

Le système ORCCAD est un environnement conçu pour spécifier, valider avec des méthodes formelles, simuler et implémenter des applications robotiques. Dans ce cadre, une application est perçue comme la composition structurée et hiérachiques en *Tâches Robot* ou TRS et en *Procédures Robot* ou PRS. Les TRS sont les actions robotiques appelées, qui combinent des aspects comportementaux mais surtout des aspects liés à la dynamique du système qu'elles contrôlent, et les PRS servent à structurer de manière logique plusieurs TRS dans le but de réaliser un objectif global, les PRS adressent donc principalement les aspects réactifs du système.

Chapitre 5

Mise en Oeuvre d'un Prototype

Tout au long de ce travail, on s'est basé sur l'exemple d'un système de vision active détectant des objets, comme décrit dans [5]. En fait, cet exemple a été l'objet de la conception d'un prototype durant ce stage, afin de servir de tremplin pour dégager des méthodologies caractéristiques qui répondent aux besoins exprimés au niveau de l'élaboration d'environnement tel que IUE mais supportant le temps-réel.

Dans ce chapitre, nous allons montrer pourquoi il est intéressant, dans un système tel que ceux qui sont visés ici, d'utiliser la norme POSIX lors de la phase d'implémentation du système. Ensuite, nous présenterons le prototype qui a été élaboré lors de ce stage et qui regroupe les aspects de réactivité synchrone avec l'utilisation du langage ESTEREL, d'interface utilisateur graphique, et de fonctionnalités compatibles POSIX. Et enfin, nous discuterons des aspects théoriques qui émergent de cette implémentation.

5.1 La Portabilité Temps-Réel

Une fois les spécifications réalisées, il faut ensuite implémenter ces spécifications. Or, pour pouvoir être intéressantes pour les milieux industriels, nos applications doivent pouvoir tourner sur n'importe quel système supportant le temps-réel (UNIX étendu, VXWORKS, etc...). Et pour ce faire, il faut que les primitives faisant appel à la gestion de ce temps-réel (gestion du temps qui n'est pas faite au niveau du noyau réactif, des timers, de la mémoire, etc...) soient compatibles d'un système à un autre. Par exemple, si on a besoin de créer un compteur, la fonction utilisée pour cela, doit pouvoir être comprise de la même manière et ceci indépendamment du système supportant le temps-réel choisi, même si au niveau de l'implémentation même de cette fonction les systèmes adoptent des choix différents, selon les caractéristiques de leur architecture.

Pour atteindre cette portabilité au niveau des systèmes temps-réel, une norme s'est créée : POSIX ou **P**ortable **O**perating **S**ystem **I**nterface [16]

C'est une norme, standardisée par *ANSI* et *ISO*. Elle propose la spécification d'un ensemble de fonctions permettant de solliciter les services de base d'un système d'exploitation. L'objectif est de permettre le développement d'applications portables au niveau du code source.

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 de rendre portables, des applications écrites pour un système d'exploitation donné, et de rendre l'écriture d'un programme plus simple. Ainsi dans les systèmes tels que SOLARIS qui sont conformes à une version de POSIX, des bibliothèques sont fournies pour permettre d'utiliser la couche POSIX. Cela se traduit comme suit :

- **au niveau de la compilation** :
  Les systèmes POSIX sont censés supporter un langage standard : le plus souvent le C-ANSI. Donc, il faut que le compilateur de l'application en comprennent les facilités pour utiliser la couche POSIX.
  L'application qui souhaite utiliser des fonctionnalités POSIX, devra faire appel, dans son code source, aux fonctions compatibles avec POSIX, qui sont regroupées dans les fichiers comprenant les entêtes des fonctions POSIX.

- **au niveau des fichiers d'entêtes** :
  Ces fichiers comprennent un ensemble d'entêtes qui définit l'interface POSIX supporté par le système. Dès que l'on a besoin d'une entête particulière dans une application, on inclue dans le code de l'application le fichier qui convient.

- **au niveau des librairies** :
  Les librairies contiennent les objets qui implémentent les fonctionnalités Posix. A l'édition de lien, ces librairies sont à lier à l'application ou à l'exécution si les librairies sont partagées dynamiquement.

- **au niveau de l'exécution** :
  Une fois l'application compilée et traitée par l'éditeur de lien, l'exécutable pourra être chargé sur une machine temps-réel supportant les fonctionnalités Posix. Ainsi on peut distinguer l'environnement de compilation et de celui de son exécution. Ceci est très intéressant car, dans la plupart des cas, les applications sont d'abord développées sur des plateformes expérimentales, et sont ensuite transférées vers les milieux industriels.

Voici quelques exemples de fichiers d'entêtes Posix :

– le fichier `<unistd.h>` : c'est le fichier d'entêtes qui définit la conformité Posix. Dans ce fichier, on trouve la définition de constantes ou de macros optionnelles que comprend un système Posix. On y trouve la déclaration des constantes `NULL`, `POSIX_VERSION` ou encore `_POSIX_TIMERS`.

– le fichier `<signal.h>` : c'est le fichier d'entêtes qui sert à décrire les signaux Posix. Ce fichier existe déjà dans le système standard Unix. Dans ce fichier, on trouve la définition
  - des signaux Posix tel que `SIGALRM, SIGKILL` ;
  - des structures de données nécessaire à la gestion des différents types de signaux comme `strut sigevent` ;
  - les fonctions qui gèrent les signaux comme `int kill(pid_t, int) ;`.

5.2 Le prototype ArGiMoGe

Le prototype *ArGiMoGe* qui a été mis-en-oeuvre s'inspire du travail effectué par [5] sur 2 caméras et dont le but était de :

– Faire de l'observation passive d'une scène avec stabilisation d'image ;

– Faire de la détection de cible.

De manière plus effective que cette étude très fondamentale, le prototype que l'on a conçu gère un système à une caméra, portant le nom de système *Argès*, et fournit à l'utilisateur une interface graphique à partir de laquelle il peut piloter le système de caméra, lancer des requêtes telles que l'acquisition d'une image ou la détection de mouvement dans la scène perçue à travers la caméra.

Ce prototype fait interagir ensemble les différents modules logiciels suivants :

- un programme ESTEREL, qui va servir d'agent de contrôle et de synchronisation;
- des fonctions du standard Posix concernant la gestion des compteurs et des séquenceurs, pour la portabilité au niveau du système temps-réel ;
- une interface utilisateur, à partir de laquelle l'utilisateur va lancer des requêtes qui seront transmises à l'agent Esterel ;
- un robot - en l'occurrence, ici, une caméra mobile et contrôlable - qui est piloté par l'utilisateur à travers l'interface utilisateur et contrôlé par l'agent Esterel ;
- des *straight-line programs*, qui ont été définis dans le chapitre 3 au paragraphe 3.3.1 et qui sont dans le cas du prototype *ArGiMoGe* des fonctions de traitement d'images contrôlées par l'agent Esterel. Ces fonctions sont des fonctions C standard qui appartiennent à une des librairies de traitement d'images du projet *RobotVis*.

Le prototype élaboré porte le nom de *ArGiMoGe*, comme contraction entre le système de caméra employé, *Argès*, et *ReViMoGe*.

5.2.1 L'interface Utilisateur

Du point de vue Utilisateur

Le prototype *ArGiMoGe* se présente comme une interface graphique homme–machine, grâce auquel un utilisateur peut lancer quatre types de procédures :

- le pilotage du système *Argès*. L'utilisateur a à sa disposition :
  - le bouton `"Acqui"` : qui lance une acquisition vidéo qui va s'afficher sur la fenêtre de gauche de l'interface ;
  - le bouton `"Init"` : qui lance la procédure d'initialisation du système *Argès*;
  - une ligne de commande `"arges >:"` situé en bas de la fenêtre d'acquisition, qui permet à l'utilisateur de taper directement les commandes pour *Argès*.
- la procédure de détection de mouvement. Pour enclencher cette procédure de détection de mouvement dans l'image reçue par la caméra, l'utilisateur clique à la souris le bouton noté "Motion Detection". Une fois cliqué, ce bouton change de couleur et est noté "Stop Detect". Ainsi, pour stopper la procédure de détection, l'utilisateur a juste besoin de recliquer sur ce bouton.
  Pour cette procédure, l'utilisateur a deux paramètres à initialiser:
  - "SmoothWin", qui correspond à la fenêtre de lissage désiré, qui va servir lors du processus permettant d'éliminer le bruit dans l'image à traiter ;
  - "GradThres", qui correspond au seuil limite du gradient à employer sur l'image lors du processus de détection de mouvement.

  Il est à noter que le positionnement de ces paramètres peut se faire tout au long du processus de détection, car toutes les actions sont traitées en temps-réel. Et bien que, dans cette version du prototype, le positionnement des paramètres "SmoothWin" et "GradThres" se fasse à la main, ces paramètres peuvent être pilotés et adaptés par des routines automatiques comme il est discuté dans [5].
- faire appel à un fichier d'aide : en cliquant sur le bouton "Help". Une fois ce bouton cliqué, l'application NETSCAPE est lancée et la page d'aide apparaît sous forme d'une page WWW.
- quitter l'application : ceci est fait en cliquant sur le bouton "Quit".

On rappelle que toutes les actions lancées par l'utilisateur sont traitées en temps–réel dans le prototype.

Voici une illustration de cette interface par la figure fig.5.1

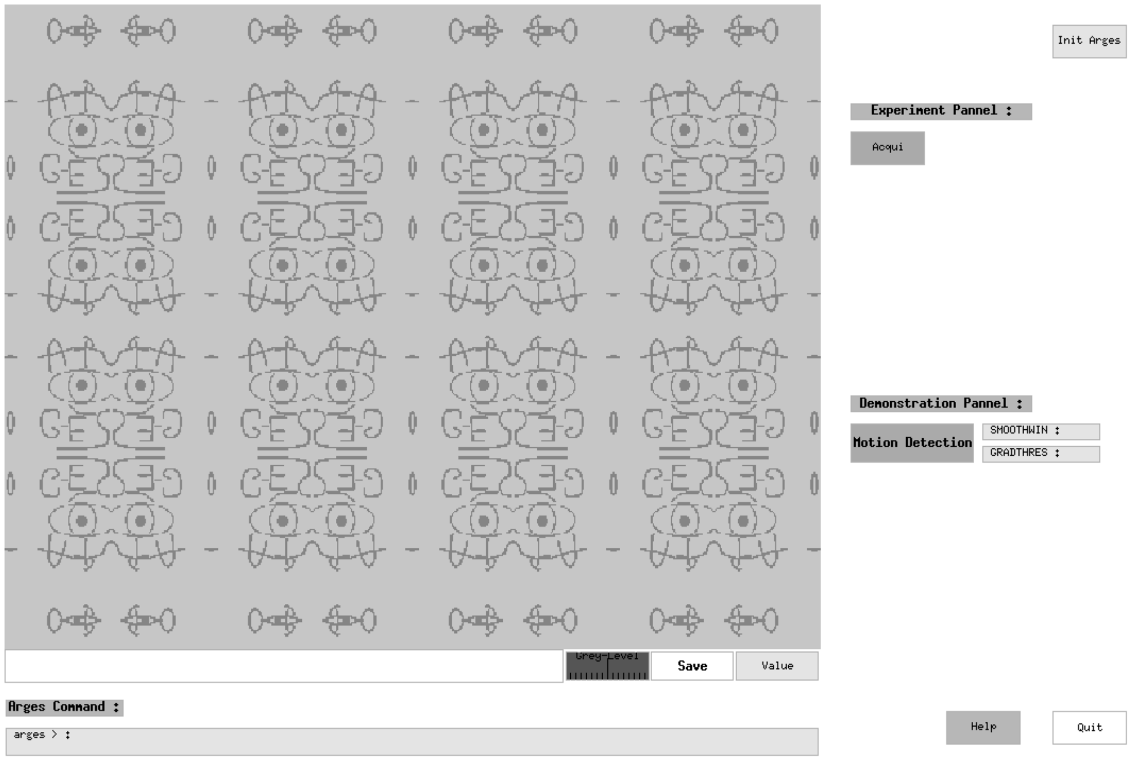

Figure 5.1: Interface du Prototype Argimoge

Du point de vue l'implémentation

Cette interface a été créée en utilisant la librairie *ActiveWindows* développée au sein du projet *RobotVis*, et qui permet de programmer des interfaces XWINDOW. Cette interface migrera sous JAVA dans un proche avenir.

Sans rentrer dans les détails de l'implémentation, chaque bouton créé est associé à une fonction *call back* qui implémente les actions à réaliser. Et dans le cas de ce prototype, chaque *call back* va faire appel au programme de contrôle ESTEREL qui décide, en fonction de l'état de l'automate, quelle fonction lancer en fonction de la fonction *call back* appelée.

5.2.2 Hardware Utilisé

Au niveau hardware, le prototype *ArGiMoGe* utilise le système monoculaire *Argès*. C'est une plateforme expérimentale développée dans le projet *RobotVis* pour tester les algorithmes de vision active 3D.

Le système *Argès* est formé 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 *cm* sur lequel peut se déplacer la tourelle Pan-Tilt.
- un panneau d'acquisition SunVideo basé sur la librairie xil.

Les figures ci-dessous montrent à quoi ressemble *Argès* :

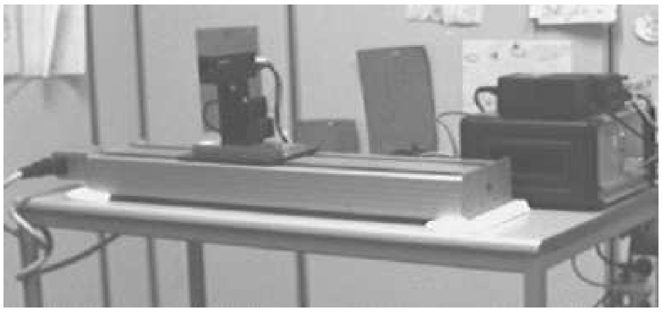

Figure 5.2: Le Système Argès

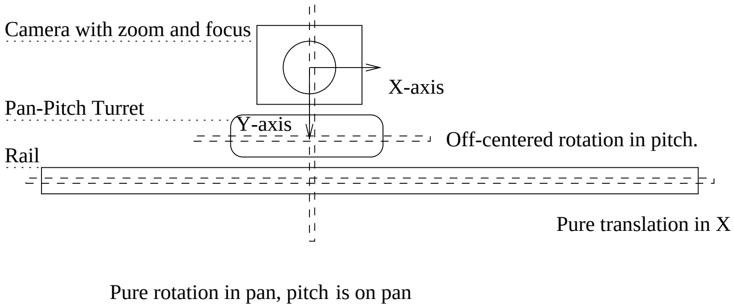

Figure 5.3: Schéma de Fonctionnement d'Argès

Le système est piloté à travers un driver logiciel défini dans le fichier d'interface `arges.h`. 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.

5.2.3 Implémentation du Prototype

Comme nous l'avons dit dans le paragraphe 5.2 de ce chapitre, le prototype tourne en temps-réel et est constitué d'un module de contrôle implémenté en ESTEREL qui a pour mission de décider et de contrôler les actions à lancer en fonction des requêtes faites par l'utilisateur à travers l'interface graphique. Le module de contrôle doit superviser principalement un système robotique *Argès* et des modules de traitement d'images.

La figure 5.4 résume en image l'architecture du prototype à l'exécution.

Fonctionnalités POSIX **:**

Les contraintes de temps que le prototype a à gérer sont de l'ordre de la cadence vidéo du système de caméra, c'est-à-dire de l'ordre de 5Hz.

L'utilisation de POSIX, dans le prototype, apparaît dans la gestion d'un timer qui permet de contrôler la durée déxécution des fonctions qui font du traitement d'image.

Le module réactif ESTEREL seul ne permet pas de gérer directement le temps. Il faut que ce soit le système qui lui fournisse un signal correspondant à un tick d'horloge par exemple.

Par conséquent, on va gérer, avec des primitives compatibles avec le standard POSIX, un timer qui aura pour but de fournir un tick d'horloge toutes les millisecondes à une fonction qui contrôle la durée d'un traitement d'image.

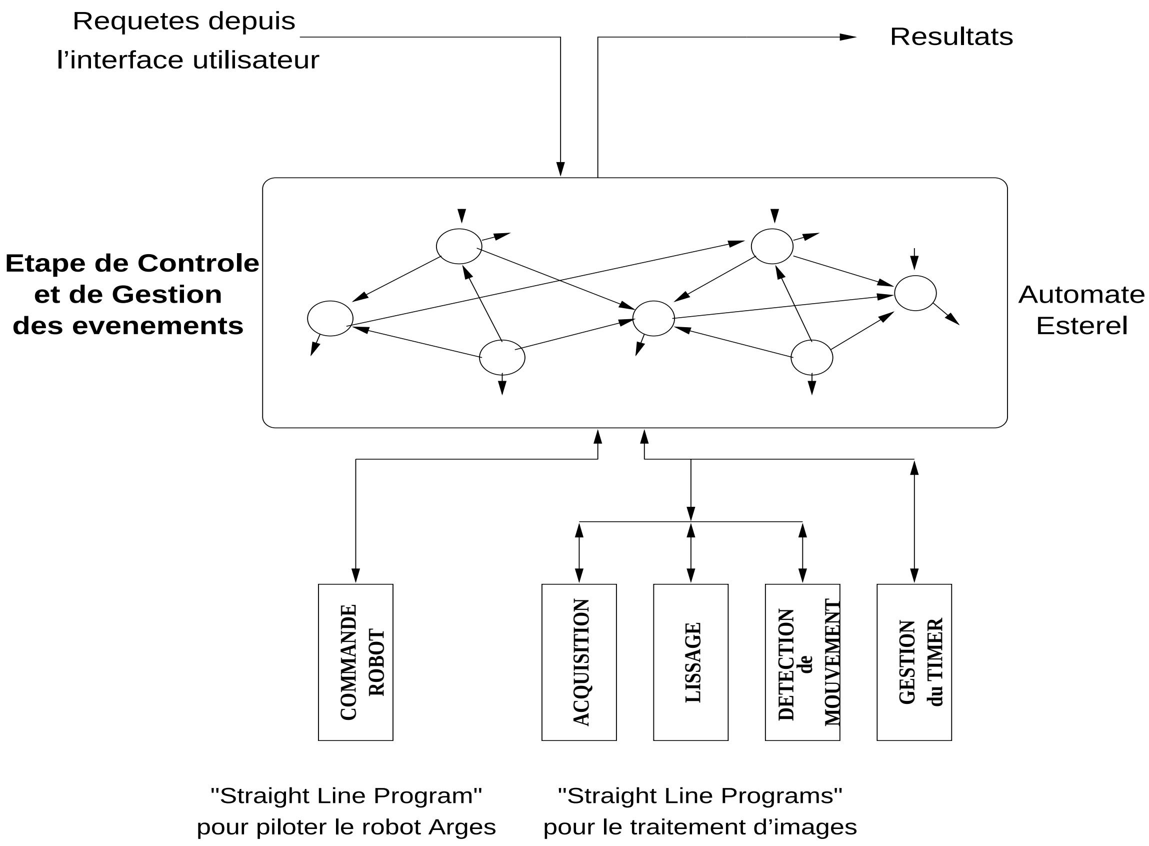

Figure 5.4: Architecture du système à l'exécution

Le protocole d'utilisation du timer suivi dans le prototype :

1. Lors d'un appel à une fonction de traitement d'image, le timer, générateur de ticks d'horloge, est enclenché par le module ESTEREL à travers le mécanisme ESTEREL du `call`.

2. Si l'action se termine dans le délai de temps qui lui est imparti, le module ESTEREL envoie un signal pour stopper le timer.

3. Sinon, au bout du laps de temps imparti à l'action à surveiller, la fonction qui gère la durée du traitement d'image stoppe le timer et envoie au module ESTEREL un signal lui signifiant que le délai s'est écoulé. Alors le module ESTEREL décide d'arrêter ou de relancer le traitement en cours.

Pour implémenter la gestion du timer, générateur de ticks horloge, on va employer les primitives compatibles POSIX suivantes :

- `int timer_create(···)` : cette primitive permet de créer le timer. On spécifie, ici, sur quelle interruption le timer va s'enclencher lors de son utilisation. De manière générale, on utilise le signal d'horloge du système.

- `int timer_delete(···)` : cette primitive permet de désamorcer le timer.

- `int timer_settime(···)` : cette primitive permet de positionner le timer avec en paramètre un intervalle de temps au bout duquel le timer expire et la fonction à appeler lors de l'expiration.

A partir de ces fonctions, deux fonctionnalités ont été créées : une pour lancer automatiquement un timer de manière périodique, en l'occurence toutes les millisecondes et l'autre pour arrêter ce timer.

Le Contrôleur ESTÉREL et l'Interfaçage C :

Tout programme ESTEREL nécessitant un interfaçage avec un autre langage, celui-ci a été fait en C -ANSI.

Le module ESTEREL doit gérer les requêtes de l'utilisateur, fournies à travers l'interface utilisateur. Ces requêtes ont été décomposées en trois types :

- la requête pour sortir de l'application ;

- les requêtes de pilotage du robot ;

- les requêtes de traitement d'images.

Donc, le module ESTEREL doit superviser :

- la durée d'exécution des fonctions de traitement d'images, à travers l'enclenchement d'un timer, comme on l'a vu dans le paragraphe précédent.

- les requêtes de pilotage du robot : si il n'y a pas de traitement en cours, le traitement correspondant est lancé par le module ESTEREL à travers le mécanisme `task-exec`, qui permet de lancer des traitements asynchrones.

- les requêtes de traitement d'images : ces requêtes sont l'acquisition d'image venant de la caméra, et la détection de mouvement, qui fait appel à 2 fonctions qui vont permettre respectivement le lissage de l'image à traiter et la détection d'une zone en mouvement. Là aussi, le mécanisme employé est celui qui fait appel à la primitive ESTEREL `exec`.
  Mais, ce traitement est encadré par un mécanisme de "chien de garde" ESTEREL, `do-watchdog`, en attente sur l'expiration du délai imparti à la fonction de traitement d'images lancée. Si le délai est expiré et que l'action de traitement d'image est en cours, celle-ci est stoppé via le mécanisme `task-exec`.

- la requête de sortie de l'application, et des signaux indiquant si il y a un problème au niveau du hardware, dans tous ces cas de figure, on sort de l'application.

La figure 5.5, comme illustration au module de contrôle ESTEREL, présente la partie de l'automate, généré par le compilateur ESTEREL, qui contrôle la requête d'acquisition d'image.

Du point de vue de l'interfaçage C, une procédure générique, dans le sens qu'elle traite toutes les commandes robot indépendamment de leur nature, lance au système *Argès* la commande adéquate à la requête de départ de l'utilisateur, et trois procédures prennent en charge respectivement le lancement de l'acquisition d'image provenant de la caméra, le lissage de l'image et la détection d'une zone en mouvement.

D'après le formalisme introduit dans le chapitre 3 au paragraphe 3.3.1, chacune de ces procédures peut être vue comme une *straight-line function*.

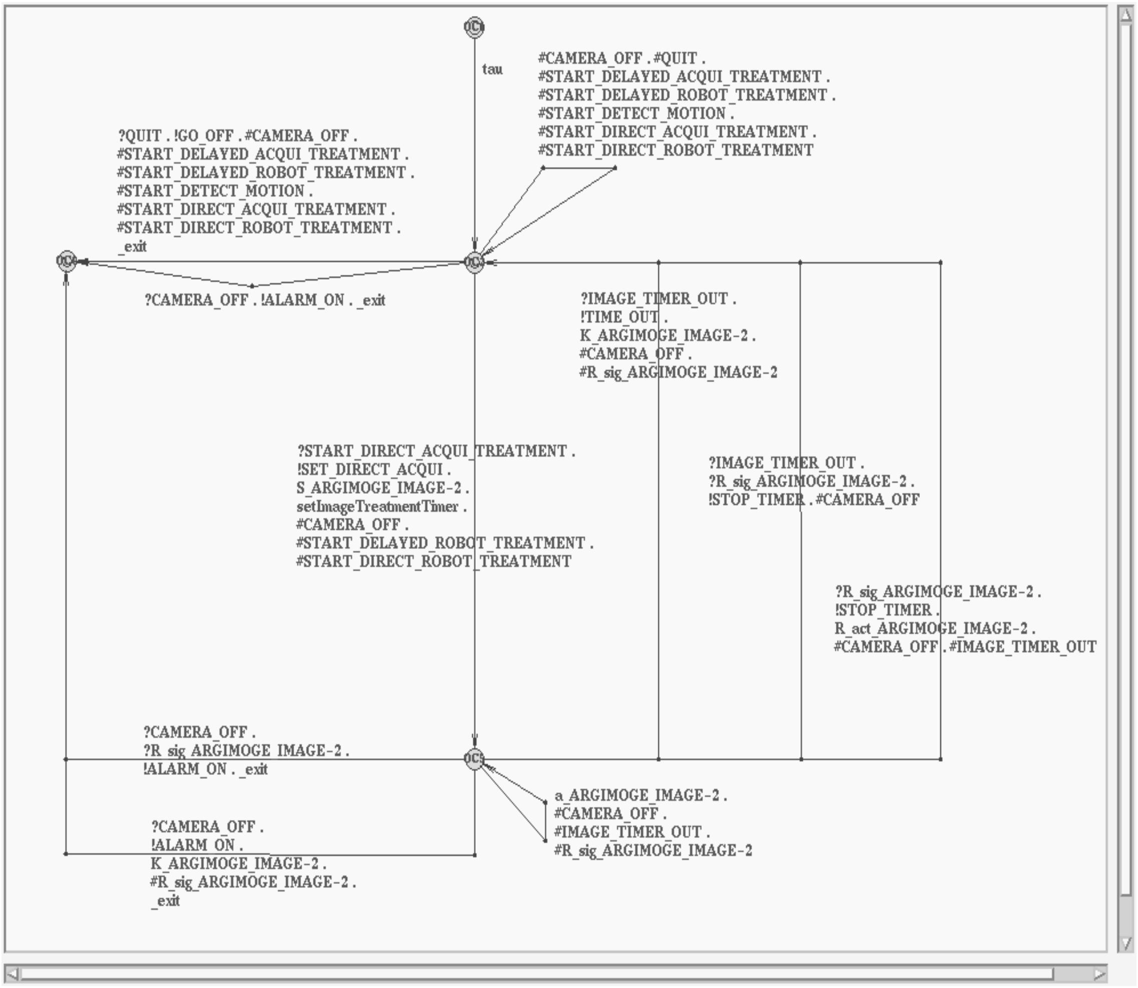

Figure 5.5: Partie de l'automate généré par ESTEREL

5.2.4 Analyse du Prototype et Perspectives :

Dans le prototype que nous avons créé, il n'a pas été question de validation. Or, c'est un point majeur dont il faut tenir compte, surtout lorsque les applications sont complexes. Par exemple, le programme ESTEREL pourra avoir un nombre d'états très importants à gérer et par conséquent deviendra très lourd à gérer pour le programmeur, du fait du grand nombre d'entités qui vont interagir.

Il paraît donc nécessaire de restreindre cette programmation en ESTEREL afin de pouvoir permettre une gestion moins lourde des programmes en permettant l'automatisation de la génération de code ESTEREL mais aussi une validation plus facile.

C'est pour cela qu'il nous est apparu intéressant d'utiliser une méthodologie qui paraît bien adaptée à ces objectifs et qui a fait ses preuves dans le domaines des applications robotiques : le système ORCCAD.

Nous avons déjà fait référence à ce système, dans le chapitre 4. ORCCAD est un environnement de développement pour la spécification, la validation, par simulation et par des méthodes formelles et l'implémentation d'applications robotiques, et s'inscrit dans le cadre de systèmes réactifs synchrones. Comme nous l'avons déjà mentionné au chapitre 4, ORCCAD décompose la complexité d'une application robot en deux niveaux d'abstraction:

- la *Procédure Robot* ou PR.

- la *Tâche Robot* ou TR ;

Une TR est la spécification complète et paramétrée :

- d'une ensemble d'algorithmes qui servent à expliciter l'action robotique (la loi de commande) qui est invariant pendant la durée de l'exécution de l'action robotique.

- d'un comportement logique déterminé par un ensemble de signaux qui peuvent subvenir avant, pendant et après l'exécution de la TR.

Le langage ESTEREL est utilisé comme moyen de spécification et d'implémentation de la partie comportementale de la TR.

Pour la partie spécification des TRS, ORCCAD utilise des événements et des traitements typés pour faciliter et automatiser cette spécification du comportement de la TR. A ce niveau, on distingue 3 classes de signaux :

1. les pré-conditions qui conditionnent l'activation d'une tâche ;
2. les exceptions qui se répartissent en 3 types : les exceptions de type 1 qui nécessitent une paramétrisation pendant l'exécution de la TR courante, les exceptions de type 2 qui demandent l'arrêt de la TR courante et un appel à la PR au niveau supérieur, et enfin les exceptions de type 3 qui demandent la mise en sécurité du système.
3. les post-conditions qui signalent la bonne terminaison de la TR.

La partie implémentation des TRS est faite à travers les *Tâches Modules* ou TMS. Ces TMS sont des tâches temps réel qui implémentent de manière optimale les fonctions élémentaires qui interviennent dans les TRS.

Les PRS représentent la spécification au niveau de la structure d'un arrangement de plusieurs TRS au niveau logique et temporel, dans le but de réaliser un objectif global. Les PRS prennent en compte l'évolution de l'environnement et ses différents types de dysfonctionnement. Ainsi, ils assurent la gestion des TRS à travers le comportement logique qui résulte de la spécification des PRS.

La traduction du comportement des PRS se fait en langage ESTEREL.

Cette présentation faite, il nous paraît intéressant de pouvoir utiliser ce système en l'adaptant aux besoins des applications de vision. Par exemple, et en reprenant le formalisme introduit dans le chapitre 3, on pourra concevoir les TR comme des *mso*, et les TMS comme les *straight-line programs*.

Dans cette perspective, une coopération entre les projets *RobotVis* et *Icare* est en cours pour permettre d'introduire dans le cadre du prototype *ArGiMoGe* ces spécifications "à la ORCCAD", et ainsi montrer la faisabilité de l'adaptation du système ORCCAD aux applications de vision.

Cette possibilité d'adapter ORCCAD aux applications de vision ouvre des perspectives tout-à-fait intéressantes quand à la définition d'un environnement qui pourrait tout comme ORCCAD spécifier, valider et implémenter des applications de vision dans un contexte temps-réel et ouvre donc de riches perspectives à un travail pour un futur travail de thèse.

5.3 ArGiMoGe en Images

Dans ce paragraphe, on présente différentes images de l'interface utilisateur du prototype *ArGiMoGe* qui illustrent les requêtes suivantes :

- une acquisition d'image (voir fig.5.6) ;
- la commande robot de déplacement du pan de la caméra (voir fig.5.7) ;
- l'acquisition d'image après le déplacement de pan (voir fig.5.8) ;
- le lancement de la détection de mouvement. Dans ce cas, 2 images résultantes de cette requête sont présentées : une sans détection de zone en mouvement, l'autre avec détection de zone en mouvement (voir fig.5.9 et fig.5.10).

Figure 5.6: Acquisition d'image depuis *Argès*

Figure 5.7: Déplacement du Pan de la Caméra

Figure 5.8: Acquisition suite au Déplacement de Pan

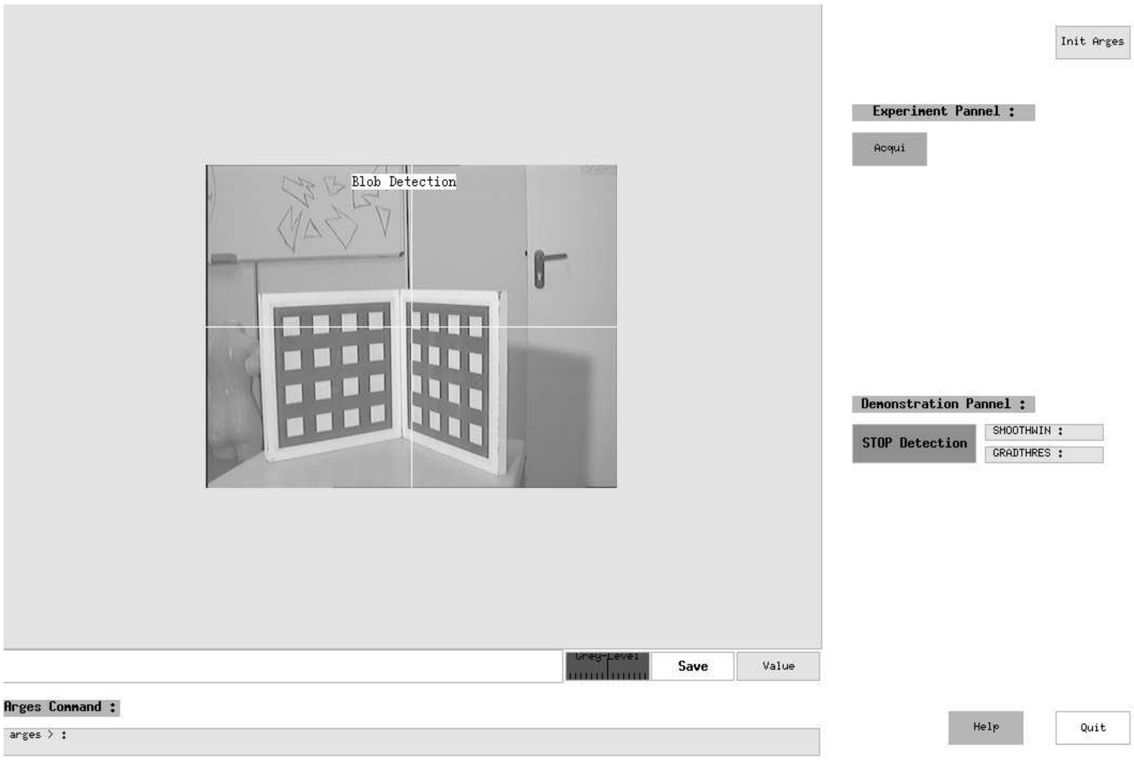

Figure 5.9: Image pendant la détection de mouvement

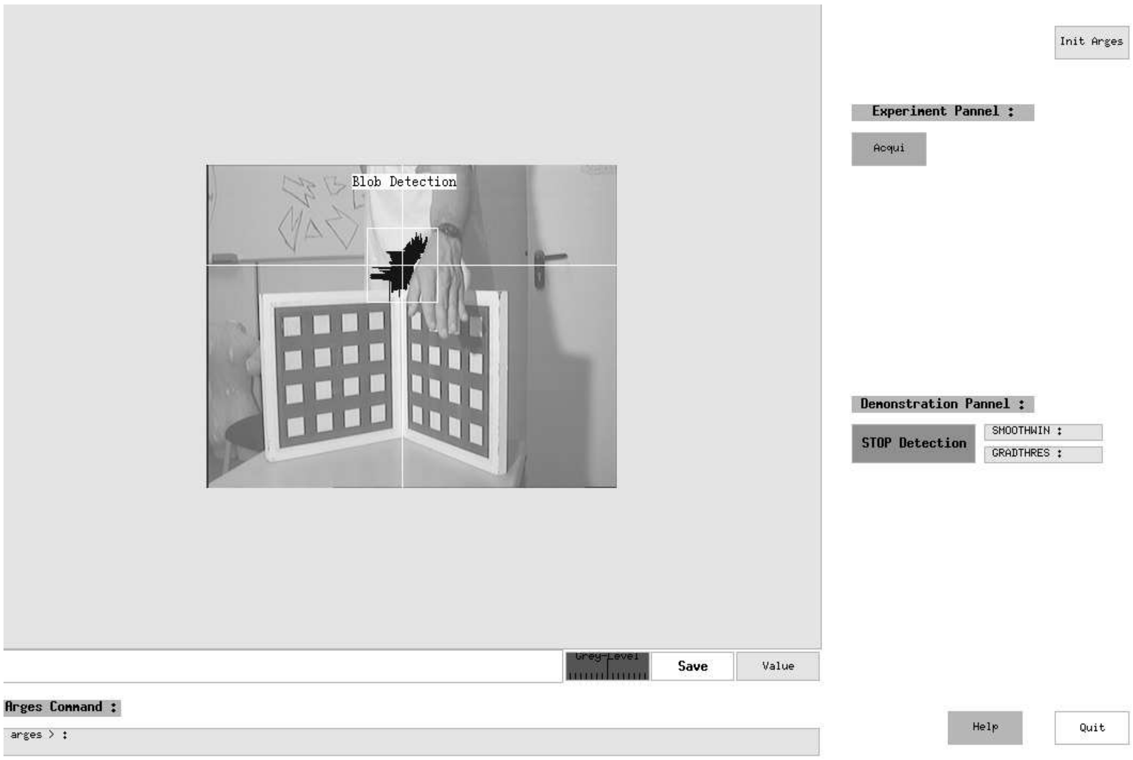

Figure 5.10: Image pendant la détection de mouvement

Chapitre 6

Conclusion

Tout au long de ce document, nous avons analysé les besoins actuels, aussi bien dans les équipes de recherche que dans le milieu industriel, et ceci de manière croissante, en matière d'environnement logiciel de programmation qui soit de la même envergure que IUE, mais qui couvrent les aspects temps réel.

Nous avons vu que l'on disposait pour ce faire, et du fait de la spécificité des modules de vision au niveau de la formalisation et des restrictions qu'ils autorisent par rapport au cas général, d'un certain nombre d'outils (ESTEREL, MAPLE, POSIX, etc $\cdots$) qui sont susceptibles d'intervenir dans l'élaboration d'un tel système.

A travers l'élaboration d'un prototype temps-réel représentatif des applications qui sont visées par le système que l'on cherche à créer, nous nous sommes efforcés de donner des réponses quant aux méthodes théoriques à utiliser pour élaborer l'environnement de programmation envisagé et quant aux outils à utiliser lors de cette élaboration, à savoir :

- une spécification orientée-objet des structures de données et des méthodes associées, grâce à son formalisme de stratification de ces spécifications objet.

- l'utilisation de l'approche réactive synchrone pour gérer le système réactif que constitue la partie comportementale du système à gérer.

- la nécessité au niveau de l'implémentation de rendre le code gérant le temps-réel le plus portable possible. Un moyen d'y parvenir est d'utiliser la norme POSIX.

Nous avons vu que le langage ESTEREL pouvait être employé pour spécifier et implémenter cette réactivité. Mais, pour obtenir un système efficace et qui puisse être valider simplement, il fallait adjoindre une méthodologie qui restreigne la programmation ESTEREL. Et un bon candidat est la méthode de programmation d'applications proposée dans le système ORCCAD, qui propose déjà un environnement de développements des applications robotiques.

Les réponses issues de ce travail ouvrent des nombreuses perspectives. A court terme, le prototype élaboré au cours du stage va être étendu pour comporter un système réactif "à la ORCCAD". Ceci permettra de montrer la faisabilité de l'adaptation du système ORCCAD aux applications de vision par ordinateur. De plus, il sera aussi intéressant d'intégrer dans le prototype : l'outil `Mascotte/mso`, développé en MAPLE.

A moyen terme, on pourrait envisager, par exemple, soit :

- étendre le système ORCCAD pour qu'il puisse gérer des missions plus complexes, à travers un planificateur qui se trouverait à un niveau d'abstraction au-dessus des *Procédures Tâches*, et qui adresserait des stratégies perceptives impliquant la planification des différents gestes perceptifs.
- parachever le formalisme dont on a donné, par ce travail, les réponses au niveau des méthodologies et des outils à utiliser. On pourrait envisager alors d'élaborer la formalisation complète d'une spécification objet stratifiée pour la génération de modules perceptifs réactifs.
- au niveau des fonctionnalités POSIX, beaucoup de travail reste encore à faire au niveau de la standardisation et de la portabilité.

Nous vous proposons finalement, une liste de pointeurs WWW où vous pourrez trouver des informations supplémentaires sur les outils ou les concepts introduits par ce travail :

- ARGÈS : `http://www.inria.fr/robotvis/personnel/vthierry/acvis-demo/arges/arges.html`
- ESTEREL : `http://cma.cma.fr:80/Esterel/esterel-fr.html`
- IUE : `http://www.inria.fr/robotvis/projects/iue/main.html`
- MASCOTTE : `http://www.inria.fr/robotvis/personnel/vthierry/mascotte/main.html`
- ORCCAD : `http://zenon.inria.fr/icare/personnel/eve/eve-publi.html`
- POSIX : `http://www.ex.ac.uk/ DCannon/WG15/WG15_sd-1.html`
- PVM : `http://rorqual.cc.metu.edu.tr:8080/sp2/pvm3/book/pvm-book.html`

- ReViMoGe : http://www.inria.fr/robotvis/personnel/sarias/revimoge.html

Bibliographie

[1] Thierry Viéville and Cyril Zeller and Luc Robert, "Using Collineations to Compute Motion and Structure in an Uncalibrated Image Sequence", *International Journal of Computer Vision*, vol. **18**, no. **2**, 1996.

[2] T. Viéville, "Mascotte : A few Maple Routines for Real-Time Code Generation", *Inria Tecnical Report*, no. **RR-2826**,1996.

[3] T. Viéville, "Using a Symbolic Calculator as a Program Generator : Application to Algorithms of Visual Perception", *Proceedings : Greco de Calcul Formel*, Luminy, Marseille 1991.

[4] F. LeGland and A Gondel, "Systematic Numerical Experiments in Non-Linear Filtering with Automatic Fortran Code Generation", *Proceedings of the 25th IEEE Conference on Decision and Control*, Athenes 1986.

[5] T. Viéville, E. Clergue, R.Enciso, H. Mathieu,"Experimenting with 3D Vision on Robot Head", *Robotics ans Autonomous Systems,Elsevier Science B.V.*, no. **14(1)**, pp 1–27, 1995.

[6] A. Benveniste and G.Berry, "The Synchronous Approach to Reactive and Real-time Systems", *Proceedings of the IEEE*, vol. **79**, no. **9**, pp 1268–1282, September 1991.

[7] F. Boussinot and R. de Simone, "The ESTEREL Language", *Proceedings of the IEEE*, vol. **79**, no. **9**, pp 1293–1303, September 1991.

[8] D. Simon, B. Espiau, E. Castillo and K. Kapellos, "Computer-Aided Design of a Generic Robot controller Handling Reactivity and Real-Time Control Issues", *IEEE Transaction on Control Systems Technology*, vol,**1**, no. **4**, pp 213–229, December 1993.

[9] E. Coste-Manière, B. Espiau and E. Rutten, "Task-Level Programming combining object-oriented Design and Synchronous Approach", *IEEE Intl Conf of Robotics and Automation*, pp 2751–2756, May 1992.

[10] B. Espiau and E. Coste-Manière, "A Synchronous Approach for Control Sequencing in Robotics Applications", *Proc. IEEE international Workshop on intelligent Motion*, pp 503–508, August 1990.

[11] B. Espiau, K. Kapellos, M. Jourdan and D. Simon, "On the Validation of Robotics Control Systems. Part I: High Level Specification and Formal Verification", *INRIA Technical Report*, **RR-2719**, submitted to IEEE Transaction on Control Systems Technology, November 1995.

[12] J. Mundy, et. al., "The Image Understanding Environnement : Overview", *Proceedings DARPA Image Understanding Workshop*, pp 283–288, April 1993.

[13] J. Mundy and T. Binford and T. Boult and A. Hanson, and R. Beveridge and R. Haralick and V. Ramesh and C. Kohl and D. Lawton and D. Morgan and K. Price and T. Strat, "The Image Understanding Environment Program", *IEEE Proc CVPR'93, New-York*, pp 406–416, June 1993.

[14] Konstantinos Kapellos, "Environnement de programmation des applications robotiques reactives", *Thèse de l'Ecole des Mines de Paris : informatique temps reel, robotique, automatique*, November 1994.

[15] Frédéric Boulanger, "Intégration de Modules Synchrones dans la Programmation par Objets", *Thèse de l'Université de Paris-Sud Orsay*, December 1993.

[16] Bill O. Gallmeister, "POSIX.4, Programming for the Real World", *O'Reilly & Associates, inc.*, 1st Edition, January 1995.

[17] Michel Beaudouin-Lafon translated by Jack Howlett, "Object-Oriented Languages", *Chapman & Hall*, 1st Edition, 1994.

[18] James Rumbaugh, Michael Blaha, Frederick Eddy, William Premerlani, William Lorensen, "Object-oriented modeling and design", *Prentice Hall*, 1991.

[19] D. Harel, A. Pnuelli, "On the Development of Reactive Systems", *Weizmann Institute of Science*, 1985.

[20] Rapháel Bernhard, Gérard Béry and cie, "Programming a reflex Game in Esterel v3", *Ecole Nationale Supérieure des Mines de Paris*, May 1989.

[21] Gérard Béry and Georges Gonthier, "The Esterel Synchronous Programming, Language : Design, Semantics, Implementation", *Ecole Nationale Supérieure des Mines de Paris*, May 1991.

[22] G. Béry, F. Mignard and J.P. Paris, "Differences between Esterel v4_41 and Previous Versions", *Ecole Nationale Supérieure des Mines de Paris*,June 1993.

[23] G. Béry, F. Mignard and J.P. Paris, "Differences between Esterel v4_45 and Previous Versions", *Ecole Nationale Supérieure des Mines de Paris*, September 1995.

[24] "Esterel V3 Documentation", *Ecole Nationale Supérieure des Mines de Paris, CISI INGENIERIES*, 1988.

Appendice A

Rappels sur les langages orientés-objet

Afin d'aider le lecteur non familier avec le vocabulaire en usage et peut-être aussi pour fixer quelques problèmes de terminologie, nous rappelons ici quelques unes des définitions des concepts se rapportant aux spécifications orientées-objet.

- **Objet ou Instance** : c'est une entité qui est capable, dans un langage orienté objet, de stocker des états et de répondre à des messages. Le terme *objet* est souvent utilisé à la place de *instance* qui souligne l'appartenance à une classe. Les *objets* ou *instances* sont créés à partir du mécanisme d'**instantiation**.

- **Classe d'objets** : le concept de classe est une extension du concept de *record* ou *struct* : une classe contient la description d'une liste de champs ($\simeq$ variables) auquel on ajoute la description d'un ensemble de méthodes ($\simeq$ fonctions ou procédures).

Exemple :

```
Pile   =        class {
       fields
                pile:  array[1..N] of integer ;
                top:  integer ;
       methods
                procedure Pop() ;
                procedure Push(value:  integer) ;
                function Top():integer ;
}
```

Il permet de définir des objets "génériques" qui seront instanciés selon les termes déja définis.

- **Héritage** : ce concept est la caractéristique propre aux langages orientés-objet. Il permet à de nouvelles classes d'être dérivées ou déduites à partir de classes déjà existantes.
  Si une classe B hérite d'une classe A, alors les instances de B contiennent les mêmes champs que ceux de la classe A, et les méthodes définies en A sont disponibles pour la classe B. Mais en plus, la classe B peut :

  – définir d'autres champs à rajouter à la liste des champs hérités de la classe A ;
  – définir de nouvelles méthodes à rajouter à la liste des méthodes héritées de la classe A ;
  – redéfinir (i.e. modifier) n'importe quelle méthode ou champ hérité de A.

  De plus, chaque méthode définie ou redéfinie en B a accès à la fois aux champs et méthodes de la classe B et A.
  On peut généraliser le processus d'héritage afin qu'une même classe hérite de plusieurs classes : c'est l'héritage multiple.

- **Encapsulation** : l'encapsulation d'objets consiste à introduire des protections pour contrôler l'accès à un objet. En général - mais cela dépend d'un langage objet à un autre -, les champs d'un objet sont privés et les méthodes sont publiques : les champs ne sont visibles qu'à travers le corps d'un objet, mais les méthodes sont visibles depuis l'extérieur.

- **Polymorphisme** : le concept de polymorphisme n'est valable que dans le cadre de langages typés, qu'ils soient ou non orientés-objet (tel que ADA ou C++), et il concerne la capacité d'un langage à décrire le comportement d'une procédure de façon indépendante de la nature (ou type) de ses paramètres. Il existe 3 types de polymorphisme :

  1. La *surchage* : on réécrit le corps de la procédure pour tous les types de données qui seront rencontrés. Le choix de la procédure à exécuter au cours du programme se fait de manière statique à la compilation en regardant le type des paramètres de la procédure. Par exemple, le C ou bien ADA utilise la surchage avec les opérateurs arithmétiques (+, -, *, /).
  2. Le polymorphisme inclusif : il est basé sur une hiérarchie partielle des types. Si le type B est sous-type de A, alors une fonction acceptant des données de type A acceptera aussi ceux de type B. Par exemple, si *char* et *booléen* sont sous-types de *entier*, alors il suffira de définir la procédure *bidon* comme : *bidon(entier,entier)* qui manipulera les 3 types définis précédemment.
  3. Le polymorphisme paramétrique : il utilise la généricité. Les types sont alors considérés comme des paramètres à la procédure générique. Par exemple, le langage ADA permet le polymorphisme paramétrique en permettant de définir des fonctions génériques.

  Pour les langages orientés-objet, le polymorphisme revêt les caractéristiques suivantes :

  * Il permet à des méthodes de 2 classes indépendantes, i.e non liées par héritage, d'avoir le même nom. Ce type de polymorphisme découle directement de la gestion des messages reçus par les instances d'objets.
  * Le polymorphisme inclusif est inhérent aux langages orientés-objet dans le sens où l'héritage introduit une notion de hiérarchie entre les classes.
  * Pour les classes liées par héritage, le polymorphisme permet de redéfinir une méthode héritée au sein d'une classe qui hérite. Cependant, ce type de polymorphisme est différent de la surchage. Et il est valable aussi dans le cas de l'héritage multiple.
  * Le polymorphisme ne s'applique pas seulement à l'instance de l'objet qui reçoit le message mais aussi aux paramètres passés dans le message.

Le message pourra accepter tout paramètre de classe héritant de la classe définie pour la méthode.

* Le polymorphisme avec héritage ne peut être résolu lors de la compilation : ce n'est qu'à l'exécution que l'on pourra déterminer la méthode à employer en fonction de la classe instanciée.

Pour plus de détails, voir [17], [18].

Appendice B

The ρ - C or Restraint C Language

B.1 Using a Restrained C-language to define processor and memory

As introduced in chapter 2, we describe, as a target language, the processor and the memory models, using *a subset of the standard C-language, called Restrained-C or* $\rho - C$. This provides us with a kind of "virtual target machine" for our specifications.

Therefore, the restrained C-language will be used for two purposes:

1. Define memory-objects, i.e. pieces of memory which contain some data on which we want to do apply pieces of code. Very restrictive aspects will be introduced, since we want:
   - To avoid any erroneous addressing, i.e. to able to demonstrate at compilation time that no memory violation or overflow wil occur.
   - To be able to compute the memory addressing time.
2. Define pieces of codes, i.e. certain sequences of instructions which modifies the value of a memory object. Similarly, very restrictive properties are required, since we want: :
   - To avoid any infinite loops or spurious jumps in the code, and to calculate the computation time, in function of the input data.
   - To be able to predict the semantic of the action performed by the piece of code.

However, it is important to note that we are not going to analyse what, in the C-language, can fit with the previous requirements and what cannot, but simply utilize a minimal set of C-features to define the memory structure and generate pieces of code.

Here the presentation of the set of C-features to be used in the *ReViMoGe* system to define the memory structure and to generate pieces of code.

B.2 ρ - C and the Memory Objects :

B.2.1 Syntax :

- Numbers :using `int` and `float` of 32 bits.
  Syntax:

$$\begin{array}{lll} number & :: & integer \mid float \\ integer & :: & \texttt{int} \quad name \; [= expr] \\ float & :: & \texttt{float} \quad name \; [= expr] \end{array}$$

- Vector, matrices or tensors : using mono-dimensional `array`s of objects. The limitation to mono-dimensional arrays is guided by the following reasons :
  - Access to multi-dimensional arrays implicitly requires multiplications for the index computation. However, most of the time, the present index can be computed from its previous value using a simple addition, depending on the algorithm.

    As a consequence the computation of these indexes must not be specified here, but at a higher level.
  - Several matrices such as symmetric matrices, diagonal matrices, triangular matrices have a special storage with a particular access function. Again this is not to be defined at this level.

  Furthermore, the index of an array -at this level of specification- can only be a constant (i.e. you can write `T[24/2]` or `T[1+3*5]`), or a bounded variable (i.e. to accept to compiling `T[i]` for an array `T[SIZE]`, the system must know that $i \in [0..SIZE[$).

  Syntax :

$$
\begin{array}{lll}
array & :: & (\, \texttt{int} \;\; | \, \texttt{float} \;\;)\; name\texttt{[}size\texttt{]} \; [\, \texttt{= \{}\; value\_list\; \texttt{\}} \,] \\
value\_list & :: & value \mid value \;\texttt{,}\; value\_list
\end{array}
$$

- Data structures : using `struct`, including *bit-fields*, and `union` of objects. This allows to entirely define the structure of a portion of memory, and the addresses at which each component can be founded.

  Syntax :

$$
\begin{array}{lll}
struct & :: & \texttt{struct \{}\; struct\_list\; \texttt{\}}\; name \\
struct\_list & :: & struct\_item \mid struct\_item \;\texttt{,}\; struct\_list \\
struct\_item & :: & (number \mid array \mid struct\_ref \mid struct \mid union \mid field) \\
 & & \\
struct\_ref & :: & struct\_name\; name\; [\, \texttt{= \{}\; \text{value\_list}\; \texttt{\}} \,] \\
value\_list & :: & value \mid value \;\texttt{,}\; value\_list \\
 & & \\
union & :: & \texttt{union \{}\; struct\_list\; \texttt{\}} \\
 & & \\
field & :: & \texttt{struct \{}\; field\_list\; \texttt{\}}\; name \\
field\_list & :: & field\_item \mid field\_item \;\texttt{,}\; field\_list \\
field\_item & :: & name \;\texttt{:}\; number \;\texttt{;}
\end{array}
$$

- Memory-objects : objects builded from these elements are called *memory objects*, they represent the structure of a part of the processor memory or the structure of a message.

  Root-Syntax:

$$memory\_object \quad :: \quad struct$$

B.2.2 Example :

A typical example of such a memory-object is :

```
struct {
 int n = 12;      /* A number, with a default value */
 union {
  float F[2*2];   /* A 2x2 matrix represented as a mono-dimensional array */
  float F00, F01,
```

```
       F10, F11; /* The same matrix represented with scalars */
 };              /* Please note that the name of the union is optional */
 other_struct s  /* Another struct defined elsewhere ... */
  = { 12, 1, 3 }, /*  ... with some initialization */
 struct {
  lsw  : 0;
  mswb : 16;
 } m;            /* A 32bits word defined as a struct of
                    two 16bits words */
} mystruct;      /* The name of the defined memory object */
```

B.2.3 Properties and Restrictions :

Obviously, such objects have the following properties:

- The memory address of an object is always known at compilation time, and *no memory violation are expected during run-time when accessing these objects.*
- The size of the object is fixed and known in advance, using the `sizeof()` operator.
- The memory mapping is the same for any implementation except for `int` and `float` which memory representation may vary from a machine to another.
- The access time to any component of a memory-object is fixed and known in advance, the address being a simple offset with respect to the memory-object start address.
- The components of the objects are always aligned with 32bits addresses. Furthermore, `struct` corresponding to bit fields are always accessed as a whole (the 32bits is read or written once and manipulated either though predefined masks or using specific processor instructions).

Let us now discuss the voluntary restrictions introduced in these specifications:

- There is no `pointer`. This is the main restriction, in fact, and the only fundamental one. As soon as pointers are introduced the semantic of the language is much more difficult to control, programs can easily violate the memory, etc...

It is clear that we **need** pointers to defined lists, tables, stacks, files etc... of elements, but these objects will not be defined this way. They will be called *containers* and will be part of another layer of the specifications and implemented in the kernel, for safe and easy use.

- There is no `double` numbers (usually of 64bits) and 8 or 16bits elements.

  This restriction is not mandatory, but seems to be reasonable considering the applications. Double floating point numbers are only useful for computations numerically not very stable, while they yield slower and heavier implementation. We thus better discourage their use, although it is very easy to reintroduce them. Similarly, using 8 and 16bits numbers save memory but very often slow the memory access since almost all architectures are now optimized for 32bits access.

  Another reason to limit the numeric formats, is to simplify the specifications and the underlying semantic models. Being lazy, helps being efficient.

- There is no `enum` or `define` primitive because this mechanism of macro is treated at another level of these specifications.

- There is also no `char` and `strings`, because the purpose of such systems is indeed not to perform text processing, but symbolic or numeric computations. Furthermore, strings are variable length zero-ended mono-dimensional arrays, which manipulation introduced pointers and unsafe access.

B.3 ρ - C and the Processor Instructions :

Let us now turns to the specification of the processor instructions we want to use. We use here a minimal set of instructions, of three kinds:

- The operators to access an array element or the component of a `struct` or an `union` :
  Syntax :

$$\begin{array}{lcl} variable & :: & variable\_array \mid variable\_struct \mid identificator \\ variable\_array & :: & variable \texttt{ [} bounded\_expression \texttt{]} \\ variable\_struct & :: & variable \texttt{ . } identificator \end{array}$$

- The Arithmetic and Logic Operators :
  - Affectation Operator : `=`

    Syntax :

    $$inst \quad :: \quad variable \texttt{ = } expr \ ;$$

  - Arithmetic Operators :

binary operator	`+, -, *, /, %`
unary operator	`-`

  - Logical Operators : they return either 1 when true or 0 when false.

<table>
<tr><td>binary operator</td><td><code>&, |</code></td></tr>
<tr><td>unary operator</td><td>!, and, or and not</td></tr>
<tr><td>cast operator</td><td><code>(int)</code> and <code>(float)</code></td></tr>
<tr><td>comparison operator</td><td>$<, >, ==, !=, <=, >=$</td></tr>
</table>

Syntax :

$$
\begin{array}{lcl}
\mathit{expr} & :: & \mathit{or\_expr} \\
\mathit{or\_expr} & :: & \mathit{or\_expr}\ \texttt{|}\ \mathit{and\_expr} \mid \mathit{and\_expr} \\
\mathit{and\_expr} & :: & \mathit{and\_expr}\ \texttt{\&}\ \mathit{eq\_expr} \mid \mathit{eq\_expr} \\
\mathit{eq\_expr} & :: & \mathit{eq\_expr}\ \texttt{==}\ \mathit{cmp\_expr} \mid \mathit{cmp\_expr} \\
\mathit{eq\_expr} & :: & \mathit{eq\_expr}\ \texttt{!=}\ \mathit{cmp\_expr} \mid \mathit{cmp\_expr} \\
\mathit{cmp\_expr} & :: & \mathit{cmp\_expr}\ \texttt{<}\ \mathit{add\_expr} \mid \mathit{add\_expr} \\
\mathit{cmp\_expr} & :: & \mathit{cmp\_expr}\ \texttt{>}\ \mathit{add\_expr} \mid \mathit{add\_expr} \\
\mathit{cmp\_expr} & :: & \mathit{cmp\_expr}\ \texttt{<=}\ \mathit{add\_expr} \mid \mathit{add\_expr} \\
\mathit{cmp\_expr} & :: & \mathit{cmp\_expr}\ \texttt{>=}\ \mathit{add\_expr} \mid \mathit{add\_expr} \\
\mathit{add\_expr} & :: & \mathit{add\_expr}\ \texttt{+}\ \mathit{mul\_expr} \mid \mathit{mul\_expr} \\
\mathit{add\_expr} & :: & \mathit{add\_expr}\ \texttt{-}\ \mathit{mul\_expr} \mid \mathit{mul\_expr} \\
\mathit{mul\_expr} & :: & \mathit{mul\_expr}\ \texttt{*}\ \mathit{cast\_expr} \mid \mathit{cast\_expr} \\
\mathit{mul\_expr} & :: & \mathit{mul\_expr}\ \texttt{/}\ \mathit{cast\_expr} \mid \mathit{cast\_expr} \\
\mathit{mul\_expr} & :: & \mathit{mul\_expr}\ \texttt{\%}\ \mathit{cast\_expr} \mid \mathit{cast\_expr} \\
\mathit{cast\_expr} & :: & \texttt{(int)}\ \mathit{uni\_expr} \mid \mathit{uni\_expr} \\
\mathit{cast\_expr} & :: & \texttt{(float)}\ \mathit{uni\_expr} \mid \mathit{uni\_expr} \\
\mathit{uni\_expr} & :: & \texttt{!}\ \mathit{cast\_expr} \\
\mathit{uni\_expr} & :: & \texttt{-}\ \mathit{cast\_expr} \\
\mathit{uni\_expr} & :: & \texttt{(}\ \mathit{expr}\ \texttt{)} \\
\mathit{uni\_expr} & :: & \mathit{variable} \mid \mathit{constant}
\end{array}
$$

Comparing to the standard C-language several aspects have been restrained here:

– No implicit cast is allowed, since -in practice- this sometimes depends on the compiler implementation, although or should not.

– When an undefined evaluation occurs, overflow, division by zero, cast to undefined value, etc.. an exception or signal will be issued, because undefined values are not acceptable here.

– Arithmetic and comparison operators apply on `int` and `float` operands, but logic operators apply on `int` operands only. Implicit casts, defined in standard-C, allows such extensions.

- The conditional, switch and particular loop instruction : Here we consider the following operators :

– conditional operator :
  `if(expr){ ... } else { ... }`

– switch operator :
  `switch name { case val1 : ··· ; ··· default: }`

– particular form of the loop operator :
  `for(int i =` a`; i` $\diamond$ b`; i = i` $\heartsuit$ c`).`

Syntax :

inst	::	*cond_inst* | *switch_inst* | *for_inst* | *seq_inst*
cond_inst	::	`if` (*expr*) `{` *inst* `}` [`else` `{` inst `}`]
switch_inst	::	`switch` (*expr*) `{` *switch_list* `}`
switch_list	::	*switch_item* | *switch_item* *switch_list*
switch_item	::	(`case` *expr* | `default`) `:` *inst* `break ;`
for_inst	::	`{ int` *i* `;`
		`for (` *i* `=` *expr* `;` *i c expr* `;` *i* `=` *i o expr* `)` [*inst*] `; }`
i	::	*identificator*
c	::	`<` | `>` | `<=` | `>=`
o	::	`+` | `-`
seq_inst	::	`{` *seq_declr* *seq_list* `}`
seq_list	::	*inst* `;` | *inst* `;` *seq_list*
seq_declr	::	| ((`int` | `float`) *name* `;` *seq_declr*

The main feature here is that we allow either conditional statements though the `if` and `switch` constructs, or a very particular kind of loop for which we can verify if it finishes and how many iterations will be performed.

B.4 ρ - C and the Straight-line Programs :

Combining these instructions, we can form sequence of expressions which affect one or more components of a memory object. These pieces of code will have several very interesting properties, because they define a specific class of programs called *straight-line programs* or `SLP` and analyzed in the sequel.

As a compilable C-object, a piece of code is defined as follows:

$$piece\_of\_code \quad :: \quad \texttt{void}\ identificator\ \texttt{()}\ seq\_inst$$

B.5 Conclusion on the ρ - C Language :

Again, several aspects of the language has been forgotten, at this stage of the specifications:

- Instructions which define general loops (for, while instruction etc...) or jumps (goto, etc...). General loops will not be treated as instructions, but using another programming mechanism, for the reason we want to control their execution.

- Some less useful operators `&&, ||, ~` are not defined since they can easily be replaced by the logical operators. Similarly, shift operators « or », which corresponds to some multiplications with eventually other mask manipulation are avoided. Furthermore, complex affectations, e.g. `+=`, `*=`, etc... or incremental `x++` operators which can be considered as compact expression builded from other defined operators are not introduced.

  Again, all these operators could have been defined, but they will not be very useful here. From a programmer point of view, the main use of these operators are the fact they might lead to more efficient compilations. This is nowadays not very true, because usual compilers indeed optimize the code even if the programmer has not written it in a very compact form. For instance, `x=x+1` will obviously be implemented, as `x+=1` or even `x++`.

- The indirect operator `*` and reference operator `&` is not to be defined since we do not manipulate pointers explicitly.

- The conditional operator `expr ? val1 : val0` is not defined, since this is redundant with the `if` instruction.

- True procedures, functions, input/output primitives, etc.. are also not defined at this stage but at another level of the specifications.

  Storage classes and other ways to qualify the data are also not used here. In fact the system will insure the coherence of all declarations using naming rules.

Finally, the target language has the following syntax:

$$
\begin{array}{lcl}
\rho - C & :: & memory\_objects \mid pieces\_of\_code \\
pieces\_of\_code & :: & piece\_of\_code \mid piece\_of\_code\ pieces\_of\_code \\
 & & \\
memory\_objects & :: & memory\_object \mid memory\_object\ memory\_objects
\end{array}
$$

which allows to define memory objects and pieces of codes.

Appendice C

A discussion on real-time implementations

In the case of continuously operating Active Vision systems, which interact with their environment (such as an industrial robot on a mobile platform), it is highly desirable that it is possible to achieve a "closed loop" of information flow. This may not be immediately achievable as a Real Time Application on a physical mobile platform, but it may be achievable in a testbed which sufficiently well simulates the real world components (environment, cameras, movement and so on).

We would like to re-emphasize this investigation into mechanisms and control structures for restrictive processing in continuous-operation vision systems. This is due to the computational demands resulting from performing robot vision in real time. It includes mechanisms for goal directed perceptual strategies such as focus of attention, fovea, and spatio-temporal scale-space, with variable space and time resolution sampling to provide sufficiently good representations of control mechanisms. The real-time restriction is interesting and important for a number of reasons:

1. Real-time vision allows for experimentation with a whole new set of applications not possible with simulations (simulations ("slowing down time") is only possible if you do not have to react to physical phenomena that have their own inherent timing).

2. Real-timing allows for creating vision applications that are responsive to the need of existing industrial settings (or in other word we can then create vision applications that respond to a world problem instead of the other way around).

3. Real-time vision is the "ultimate" version of "fast vision" (as real-time in a textbook refers to timing in fixed time-frame that is shorter than what is needed for a particular task). "Fast vision" has shown to provide at least partial answers to problems such as optical flow (due to the limitations of possible movements between frames) ...etc.

In order to be acceptable this real-time system is :

- **Theoretically well founded** :

  For example, algorithms provided by the system should have a formal specification (pre- and post-conditions). The complexity of the algorithms should be known. Data structures should also be described formally, with invariants. Concurrency and real-time control should be based on established theory, e.g. CSP.

- **Open ended** :

  There is no way that a system can provide support for every conceivable vision algorithm. Since such a system tries to cover applications, it will tend to be too large, monolithic, and hard to use and support if not modular.

  Therefore, such a system consists of a small kernel to which problem-oriented code are added as modules.

  However, it is crucial that the kernel is properly decomposed and layered, with well-defined interfaces to every part of the kernel, so that it will be possible to choose alternative implementations of one or several services in the kernel. One reason for choosing an alternative implementation is increased efficiency, another is integration and compatibility with other systems.

  For the same reason, the kernel must not contain anything that could just as well be implemented in add-on libraries. For example, support for the EUI data structures should be kept out of the kernel. For distributed systems, the naming of processes, objects and services could be implemented by a general, replaceable name service in user-space. Scheduling should be implemented a user-defined scheduler, and must not be hard-wired into the kernel. Support for the transportation and distribution of objects should be based on protocols, which allow for multiple programming languages, transport layers, and object persistence implementations.

- **Based on main stream technology :**
  Many existing systems contain deep type hierarchies with a single root. In such systems, all useful function tends to drift towards the root, whose interface quickly gets fat. The single root also makes it virtually impossible to combine the type hierarchy with other systems. In contrast, new object-oriented systems tend to contain small, shallow, orthogonal type hierarchies which are combined with multiple inheritance (mix-in techniques).

  Some systems invent their own specification languages, scripting languages, graphical interface, when there are existing, generally accepted alternatives. The use of mainstream components increases the probability that the system will be accepted and used.

  Many existing distributed image processing system (AVS, Khoros etc) are based on computational networks, consisting of filters which operate on data streams. While this abstraction might be appropriate for some image processing problems, reactive real-time systems needs more control over the computations. There is a strong trend towards distributed object-oriented systems, in which the programmer gets the feeling that all objects exists in a singe process, but where some method invocations are actually implemented by transparent remote procedure calls, shared buffers access, etc (CORBA, DOE, Spring and others).

- **Well engineered :**
  The aesthetic aspect of a system should not be underestimated. Most successful software packages are good compromises between generality, efficiency, and size. The concepts are often simple and straight-forward. The software are usually build on standard components, as far as possible, and use wide-spread, well suited programming languages.

  It is also important in order to create embedded applications where you do not need to "download" more than what is actually used. Moreover, it could enable a "progressive learning scheme" so that a newcomer does not need to learn more than is basically needed for his application to get started.

- **With support for development :**
  Several existing system lacks adequate tools for testing and debugging new applications. The system must be robust and e.g. detect crashed processes and communication problems. It must be possible to monitor program execution,

especially in distributed systems, and to trace events.

The real-time restriction is interesting and important also for a number of reasons, related industrial applications :

1. Real-time vision allows for experimentation with a whole new set of applications not possible with simulations (simulations ("slowing down time") is only possible if you do not have to react to physical phenomena that have their own inherent timing).

2. Real-timing allows for creating vision applications that are responsive to the need of existing industrial settings (or in other word we can then create vision applications that respond to a world problem instead of the other way around).

3. Real-time vision is the "ultimate" version of "fast vision" (as real-time in my textbook refers to timing in fixed time-frame that is shorter than what is needed for a particular task). "Fast vision" has shown to provide at least partial answers to problems such as optical flow (due to the limitations of possible movements between frames) etc.

In terms of building real-time systems there are several commercially available operating systems, and recently tools for specification of such systems have also become available in the academic community. It is, however, characteristic that the real-time tools available today all assume the real-time systems are homogeneous, from a complexity point of view, and typically only a few selected routines require real-time response. In fully fledged computer vision systems the need for guaranteed response times varies from a few milliseconds (low-level control loops) to several seconds (symbolic interpretation and planning tasks). The requirements, in terms of response times, are consequently highly heterogeneous. In addition computer vision and image analysis is characterized by an excessive dataflow (6-20 MB/s), this in turn requires that an execution environment must be able to create and destroy large data structures without any effect on the performance of the system. On top of this facilities for control of processing, analysis and interpretation must be supported/provided to ensure limited size models.