Contribution pour une nouvelle approche du dialogue homme-machine en C.F.A.O

Benoît Martin

THÈSE

Présentée à

l'UNIVERSITÉ de METZ

Pour l'obtention du grade de :
DOCTEUR de l'UNIVERSITÉ de METZ

Spécialité : INFORMATIQUE

Benoît MARTIN

CONTRIBUTION POUR UNE NOUVELLE APPROCHE DU DIALOGUE HOMME-MACHINE EN C.F.A.O.

Soutenue à Metz le 22 décembre 1995

Composition du jury :

Directeur de thèse :	Yvon	GARDAN	(Professeur à l'Université de Metz)
Rapporteurs :	Ileana	COSTEA	(Professeur à l'Université de Californie)
	Marie-Christine	HATON	(Professeur à l'Université de Nancy 1)
Examinateurs :	Didier	GALMICHE	(Habilité à diriger des recherches à l'Université de Nancy 1)
	Jean-Pierre	JUNG	(Professeur à l'Université de Metz)

IFORMATIQUE DE METZ

THÈSE

Présentée à

l'UNIVERSITÉ de METZ

Pour l'obtention du grade de :
DOCTEUR de l'UNIVERSITÉ de METZ

Spécialité : INFORMATIQUE

Benoît MARTIN

CONTRIBUTION POUR UNE NOUVELLE APPROCHE DU DIALOGUE HOMME-MACHINE EN C.F.A.O.

Soutenue à Metz le 22 décembre 1995

Composition du jury :

Directeur de thèse :	Yvon	GARDAN	(Professeur à l'Université de Metz)
Rapporteurs :	Ileana	COSTEA	(Professeur à l'Université de Californie)
	Marie-Christine	HATON	(Professeur à l'Université de Nancy 1)
Examinateurs :	Didier	GALMICHE	(Habilité à diriger des recherches à l'Université de Nancy 1)
	Jean-Pierre	JUNG	(Professeur à l'Université de Metz)

LABORATOIRE DE RECHERCHE EN INFORMATIQUE DE METZ

À mes parents.

« Tout ce qui ne me tue pas, me fait grandir »

« Je voudrais donner et distribuer,
jusqu'à ce que les sages parmi les hommes
soient redevenus joyeux de leur folie,
et les pauvres, heureux de leur richesse »

NIETZSCHE Friedrich (1844-1900)

REMERCIEMENTS.

Ce travail a été réalisé au Laboratoire de Recherche en Informatique de Metz, sous la direction de Monsieur le Professeur Y. GARDAN. Je tiens à lui exprimer ma reconnaissance pour la confiance qu'il m'a témoignée, pour son aide, son dynamisme et sa disponibilité dont il m'a fait profiter tout au long de cette thèse.

Qu'il me soit permis de remercier Mesdames M.-C. HATON, Professeur à l'Université de Nancy 1, et I. COSTEA, Professeur à l'Université de Californie, pour avoir accepté de rapporter ce travail et pour leur participation à la commission d'examen.

Mes remerciements vont également à Messieurs D. GALMICHE, Habilité à diriger des recherches à l'Université de Nancy 1, et J.-P. JUNG, Professeur à l'Université de Metz, pour avoir accepté de juger mon travail.

Je suis très reconnaissant à Isabelle, membre de l'équipe dialogue du L.R.I.M., pour sa collaboration et pour l'intérêt qu'elle a porté à mon travail.

Je ne saurais oublier les autres membres du laboratoire qui m'ont aidé, soit par leurs compétences ou par leur amitié.

TABLE DES MATIÈRES.

Introduction.....7

Chapitre 1. Le dialogue homme-machine.

1. Introduction.....10
2. Les modèles d'interaction.....11
3. Les styles d'interaction.....12
    3.1. Les menus.....13
    3.2. Les langages de commandes.....13
    3.3. Le langage naturel.....14
    3.4. Les formulaires.....15
    3.5. Les questions et réponses.....16
    3.6. La manipulation directe.....16
    3.7. Conclusion.....17
4. Les modèles d'architecture.....18
    4.1. Le modèle Seeheim.....18
    4.2. Le modèle IDS.....20
    4.3. Un modèle multi-agents : PAC.....21
    4.4. Conclusion.....22
5. Les modèles de dialogue.....23
    5.1. Les grammaires.....24
        5.1.1. Les grammaires hors contexte.....24
        5.1.2. Les règles de production.....26
        5.1.3. Les algèbres de processus.....27
    5.2. Les réseaux de transitions.....28
    5.3. Le modèle à événements.....31
    5.4. Les langages déclaratifs.....33
    5.5. Conclusion.....33
6. Les outils de développement.....34
    6.1. Les systèmes de gestion de fenêtres.....35
    6.2. Les boîtes à outils.....37
    6.3. Les systèmes d'aide au développement.....38
    6.4. Conclusion.....40
7. Conclusion.....41

Chapitre 2. Un modèle d'interaction pour la C.F.A.O.

1. Introduction.....43

2. La C.F.A.O. 43
3. La définition statique. 47
3.1. Les menus. 47
3.2. Les actions globales. 48
3.3. Les actions interactives. 49
3.3.1. La primitive INTERACTION. 50
3.3.2. Les opérateurs de composition. 52
4. La définition dynamique. 54
4.1. Les compatibilités. 55
4.2. La définition des compatibilités. 57
4.3. La représentation des compatibilités. 59
4.3.1. Les graphes de compatibilité. 60
4.3.2. Les arbres de compatibilité. 60
4.3.3. Synthèse. 61
4.4. L'évaluation des compatibilités. 62
4.4.1. Sélection d'un menu compatible. 62
4.4.2. Validation d'une interaction. 64
4.5. Les compatibilités explicites. 66
4.6. Les effets et les compatibilités. 66
4.7. Le différé et l'annulation. 66
5. Implantation et exemple. 67
6. Conclusion. 70

Chapitre 3. Les modèles de dialogue de SACADO.

1. Introduction 73
2. Le générateur de menus. 74
3. Vers un langage d'actions. 75
4. Un langage textuel : NADRAG. 76
4.1. Les structures de données. 77
4.2. Les instructions graphiques. 78
4.3. Les instructions du dialogue. 79
4.3.1. La primitive INTERACTION. 81
4.3.2. Les opérateurs de composition. 82
4.4. Les structures de contrôle. 84
4.5. Les actions globales. 86
4.6. Implantation et conclusion. 87
5. La programmation visuelle. 87
5.1. Un langage graphique. 88
5.2. Les entités. 90

5.2.1. Les interactions. 90
5.2.2. Les tâches. 92
5.3. Les relations. 93
5.3.1. Les effets. 93
5.3.2. Les contraintes. 94
5.3.3. Le flot de données. 94
5.3.4. La séquence. 94
5.4. Les actions globales. 96
5.5. Implantation et conclusion. 98
6. Conclusion. 99

Chapitre 4. La génération du dialogue.

1. Introduction 102
2. Le développement du dialogue. 102
3. Le modèle et les comportements. 105
3.1. Les objets. 106
3.2. Les producteurs. 107
3.3. Les transmuteurs. 109
3.3.1. Les transmuteurs intrinsèques. 110
3.3.2. Les transmuteurs extrinsèques. 111
3.4. Les propriétés des objets. 113
3.5. Les réacteurs. 116
3.5.1. Les réacteurs intrinsèques. 116
3.5.2. Les réacteurs extrinsèques 118
4. La génération du dialogue 121
4.1. Les producteurs. 122
4.2. La composition de producteurs. 125
4.3. Les transmuteurs. 126
4.4. Les propriétés des objets. 127
4.5. Les réacteurs extrinsèques. 129
5. Implémentation et conclusion. 130

Conclusion 132
Références bibliographiques 134
Annexes. Introduction 143
Annexe A. Le développement de SACADO. 147
Annexe B. Le langage NADRAG. 164
Annexe C. Les bibliothèques de SACADO 169
Annexe D. Un modèle par les comportements. 179

INTRODUCTION.

Durant ces quinze dernières années, des progrès notables ont été accomplis dans le domaine des interfaces utilisateur et notamment pour les Applications Graphiques Interactives (AGI). On est passé d'un dialogue rudimentaire à une forme de dialogue plus évoluée, où l'utilisateur interagit directement dans un espace de travail reconstitué graphiquement.

Cependant, la conception de telles interfaces devient un des problèmes clés pour le développement des AGI. Les coûts de développement deviennent critiques et la complexité des applications s'accompagne d'un accroissement de celle de l'utilisation qui est pénalisante dans la mesure où la tendance est de mettre les logiciels dans les mains d'utilisateurs non informaticiens. Ce dernier point nécessite une prise en charge de l'utilisateur à tous les niveaux, de le guider voire de l'aider à une bonne utilisation du logiciel. Cette aide ne doit pas l'empêcher de suivre sa propre logique (liberté d'agir) tant que ses actions ne sont pas incompatibles avec le logiciel. Bien entendu, ce suivi de l'utilisateur dans son interaction avec le logiciel se fait grâce à une forte interactivité à chaque fois que cela est possible.

Tout au long de ce mémoire, notre objectif est de fournir à l'utilisateur la possibilité de construire sa propre interface. Pour cela, il doit disposer de modèles suffisamment précis pour soulager son travail de conception; c'est une étape incontournable. Par modèles, nous entendons la formalisation du comportement de l'utilisateur et des différents échanges mis en jeu lors des interactions avec le logiciel. Par la prise en charge implicite d'un grand nombre de concepts, il est alors possible de créer des outils accessibles au plus grand nombre. Cependant, si cette approche a le mérite d'imposer une certaine discipline dans le développement des interfaces utilisateur (règles à respecter), ces outils ne rendent pas pour autant facile le travail de spécification. Avec la grande variété des objets manipulés par les logiciels et par les AGI en particulier, ce travail est même très long et source de nombreuses erreurs. On aboutit le plus souvent à des interfaces incomplètes par rapport aux objets manipulés. Pour répondre aux difficultés de développement et au problème de complétude, nous pensons que la sémantique très forte qui est incluse dans le modèle des objets est un point de départ primordial dans le développement des interfaces utilisateur. Par une transformation adéquate de la connaissance liée aux objets de ce modèle, il nous apparaît possible de déduire en grande partie l'interface utilisateur du logiciel. Cette déduction doit nous garantir le caractère complet de l'interface car aucune possibilité n'est oubliée. De plus, dans la mesure où cette interface déduite se présente sous le même formalisme que celui proposé à l'utilisateur, il sera toujours possible d'intervenir pour l'adapter ou l'enrichir pour des besoins particuliers. C'est cette dualité dans le développement de l'interface qui nous permet d'envisager qu'un utilisateur non informaticien puisse concevoir sa propre interface voire l'architecture complète de son logiciel.

Le chapitre un montre que de nombreux modèles ont été introduits pour aider à la prise en charge de l'utilisateur et au développement des interfaces utilisateur. Les modèles d'architecture proposent une décomposition statique du logiciel en composants indépendants. Tous ces modèles ont comme point commun de séparer au maximum le dialogue et l'application proprement dite. Les modèles

d'interaction formalisent le comportement de l'utilisateur lors d'un échange avec le logiciel. On associe le plus souvent à ces modèles, ce que l'on appelle le style d'interaction, qui décrit la forme que prend l'interaction. Pour le développement, les modèles de dialogue fournissent des méthodes pour spécifier la dynamique des échanges entre l'utilisateur et le logiciel indépendamment de toute notion d'implémentation. Nous proposons une étude de ces différents modèles et des outils de développement couramment rencontrés avant d'aborder notre domaine d'application, la C.F.A.O. (Conception et Fabrication Assistées par Ordinateur).

S'il est un domaine où l'interface utilisateur et notamment les interfaces graphiques sont primordiales c'est celui de la C.F.A.O. L'utilisateur interagit avec un modèle des objets pour construire une maquette virtuelle de l'objet qu'il désire manipuler. De tels systèmes laissent une multitude de possibilités à l'utilisateur et engendrent des résultats complexes. Après une étude des difficultés propres à la C.F.A.O. et de ses concepts généraux, nous proposons dans le deuxième chapitre une formalisation précise du modèle d'interaction du système SACADO (Système Adaptatif de Conception et d'Aide au Développement par Ordinateur) développé au Laboratoire de Recherche en Informatique de Metz et dont l'étude a débuté en 1986. Ce modèle prend en compte les intentions de l'utilisateur à travers une primitive unique de dialogue appelée INTERACTION. Elle assure une grande liberté à l'utilisateur tout en garantissant la cohérence des dialogues engagés.

Cette primitive unique ouvre tout naturellement la voie au développement du logiciel autour du dialogue et autour de l'interaction en particulier. Dans le troisième chapitre, nous étudions deux modèles de dialogue (langages d'actions) basés sur notre modèle d'interaction : le langage textuel NADRAG et un langage (formalisme) graphique. Nous présentons tout d'abord l'intérêt de tels modèles avant d'en détailler les caractéristiques.

Toutefois, il nous apparaît essentiel d'aller plus loin dans l'aide apportée au développement grâce à une automatisation duale. À partir des connaissances incluses dans le modèle manipulé par l'utilisateur, nous proposons de générer le dialogue sous la forme du langage graphique abordé précédemment. En utilisant les travaux réalisés sur ce langage graphique, il est alors aisé de préciser et de modifier les dialogues générés et créés. Le quatrième chapitre décrit le modèle des objets que nous envisageons (basé sur des comportements) et met en avant les différentes générations proposées.

Les travaux de ce mémoire ont fait l'objet de nombreuses implémentations dont les descriptions se trouvent dans les quatre annexes.

CHAPITRE 1.

LE DIALOGUE HOMME-MACHINE.

1. INTRODUCTION.

L'outil informatique doit permettre à une personne d'interagir avec d'énormes quantités de données associées à un grand nombre de fonctions. Le contrôle des commandes de l'utilisateur et des réponses du système définissent une interface, un échange d'informations entre ces deux intervenants [MAR 91]. Très souvent, la qualité d'un logiciel dépend de la qualité de son interface et en particulier de son adaptation en fonction de l'utilisateur, qu'il soit novice ou expérimenté [FOL 90; NEE 91]. Dans des conditions particulières, une interface dont la conception n'est pas adaptée à l'utilisateur peut entraîner des catastrophes : par exemple une mauvaise ergonomie concernant les commandes de vol d'un avion peut entraîner de fausses interprétations de la part du pilote. Il convient donc de prendre en compte les capacités de l'utilisateur potentiel afin d'équilibrer son effort et son travail pour atteindre son objectif [HAN 88].

Le développement des interfaces a toujours été une tâche difficile [BAE 95]. Leur complexité ne cesse d'augmenter en parallèle avec celle des applications, notamment les Applications Graphiques Interactives (AGI) [MAR 92a]. Une AGI est un outil qui produit des dessins ou des images sur une surface de visualisation et qui est destiné à interagir graphiquement avec l'utilisateur [GUI 93]. Ainsi, ces applications doivent maintenir constamment un affichage cohérent des données. Ceci nécessite de très bonnes performances pour assurer qu'il n'y a pas de retard perceptible entre les actions de l'utilisateur et la réponse du système. Le code ainsi produit et dévolu à l'interface est souvent très important, complexe et difficile à gérer, sa taille pouvant même dépasser celle de l'application. Généralement, plus l'interface est facile à utiliser, plus il est difficile de l'écrire. Les équipes qui interviennent dans de tels développements sont pluridisciplinaires, ce qui nécessite de fédérer les intervenants qui sont des graphistes, des ergonomes, des spécialistes des sciences cognitives, ... [MAR 93].

De manière conventionnelle, la conception est faite avant que l'implémentation ne commence, la conception forme alors une base solide pour l'implémentation. Cependant, encore très souvent, la seule façon de développer des interfaces de qualité est de tester des prototypes avec des utilisateurs et de modifier la conception en tenant compte de leurs remarques. En effet, il n'existe pas de règles ou de techniques systématiques pour garantir qu'un logiciel sera facile à utiliser [MYE 89; SIN 90]. De nombreux prototypes sont nécessaires et ils sont très souvent modifiés. La demande en outils de développement pour aider à concevoir et à implémenter des interfaces se fait de plus en plus forte. L'objectif de ces outils, au delà de l'utilisation des dernières avancées technologiques et matérielles, est d'avoir des logiciels plus faciles à appréhender et à utiliser mais aussi à produire et à maintenir [PAN 93].

Dans ce chapitre, nous nous intéressons à la problématique du développement des interfaces homme-machine. Pour cela, nous définissons ce que l'on appelle les modèles d'interaction qui caractérisent la communication entre l'utilisateur et la machine ie. l'interaction homme-machine. Nous présentons les différentes formes que peut prendre cette interaction, les styles d'interaction, en faisant ressortir leurs

domaines d'utilisation. Enfin, nous décrivons les modèles les plus fréquemment utilisés pour manipuler et spécifier ces interactions. Il s'agit aussi bien des modèles d'architecture, qui proposent une structure pour décomposer un système, que des modèles de dialogue qui donnent des méthodes pour spécifier la structure du dialogue homme-machine.

2. LES MODÈLES D'INTERACTION.

On définit généralement l'interaction par la communication établie entre un utilisateur et le système. De nombreux modèles ont été introduits pour formaliser cet échange d'informations. Le modèle de Norman est certainement le plus utilisé [NOR 86]. Il est basé sur la répétition de cycles évaluation-exécution et correspond très bien à l'idée intuitive que l'on a d'une interaction. L'utilisateur construit un plan d'actions qui est exécuté par l'interface. Lorsque ce plan est exécuté en tout ou partie, l'utilisateur observe l'interface pour évaluer le résultat et déterminer les actions suivantes. La construction du plan d'actions se caractérise par la distance d'exécution qui mesure la distance entre ce que veut faire l'utilisateur et comment le faire avec le système. L'évaluation du résultat de l'exécution de ce plan d'actions se définit par la distance d'évaluation qui mesure la distance entre ce qui est affiché et ce que l'utilisateur interprète.

Ainsi, la difficulté essentielle de l'interaction provient des différences profondes qui existent entre les deux interlocuteurs. Les deux distances présentées en sont des exemples. Les caractéristiques de l'utilisateur sont très différentes : il est par définition adaptable et créatif. Sa capacité d'adaptation lui permet de comprendre ce que la machine fait et de réagir en conséquence. Son côté créatif implique qu'il est impossible de décrire de façon exhaustive tous les comportements possibles. Une interaction doit lui permettre de suivre sa propre logique tout en respectant les impératifs du logiciel. Ce point est un problème central dans le développement des interfaces homme-machine.

Contrairement aux langages traditionnels où la syntaxe est stricte et non contextuelle, un langage d'interaction doit prendre en compte le caractère opportuniste de l'utilisateur qui n'atteint pas son but de façon rectiligne [COU 90]. Un tel langage doit posséder une syntaxe permissive et largement contextuelle. Plus l'interaction est souple, moins l'utilisateur doit s'adapter au système pour entrer ses requêtes. Les dialogues non modaux définissent des interactions indépendantes les unes des autres. L'utilisateur peut interrompre un dialogue pour un autre ou revenir au premier. Ainsi, l'utilisateur est libre et ne se sent pas prisonnier d'un dialogue dont il ne sait comment en sortir. On parle alors de fils d'activités multiples. Cela répond à la démarche basée sur l'essai et l'erreur qui caractérise le processus de résolution des problèmes souvent appliqué par l'utilisateur. Lorsque plusieurs dialogues sont pris en compte simultanément, on parle même de dialogues concurrents : par exemple le déplacement d'un objet tout en l'aggrandissant. Par opposition, les dialogues modaux sont basés sur des interactions de type conversationnel dont on ne peut sortir que par les réponses prévues.

Le dialogue d'une interface est défini par l'ensemble des interactions qui le composent. Le séquencement ou le déclenchement de chaque interaction se fait selon des règles établies et sous un certain contrôle. Ce contrôle détermine le déclenchement d'une interaction. On distingue deux types de

contrôle :

- interne;
- externe.

Le contrôle est interne lorsque l'interaction est contrôlée et dirigée par l'application. L'application appelle simplement les fonctions d'entrée de l'interface pour obtenir la donnée appropriée. Cette méthode est généralement utilisée lorsque les données à manipuler possèdent une très forte sémantique mais elle noye le dialogue dans l'application ce qui rend très difficile sa mise au point et sa modification. De plus, cela ne favorise pas l'indépendance entre le dialogue et l'application.

Le contrôle est externe lorsque l'interface dirige les interactions et appelle l'application en réponse aux actions et aux commandes de l'utilisateur. Le dialogue est spécifié en même temps que des indications sur les endroits où doivent être appelées les fonctions de l'application. Le séquencement et la planification sont sous la responsabilité du dialogue. Le flot de données est interne au dialogue et l'application est vue comme un ensemble de modules fonctionnels à appeler lorsque cela est nécessaire. Les fonctions de programmeur d'interfaces et de programmeur d'applications sont clairement distinguées.

Le choix du contrôle est très important car il fixe le niveau de découpage du système. Par un contrôle externe, le système se trouve décomposé en éléments fonctionnels de base alors qu'à l'inverse, par un contrôle interne, le système est monolithique. De plus en plus, on utilise un contrôle mixte issu des deux modes précédents, ce qui permet de choisir le niveau de découpage désiré.

3. Les styles d'interaction.

L'interaction homme-machine étant caractérisée, il est important de définir comment l'utilisateur indique à la machine la tâche à réaliser : c'est le style d'interaction [BAE 95; DIX 93; PRE 94]. Il est essentiel que les ordres inhérents à cette tâche soient bien formulés et qu'ils soient correctement interprétés par la machine. On distingue deux catégories : verbe-objets et objets-verbe. Dans le cas d'une interaction verbe-objets, le dialogue se déroule par requêtes, chaque commande est suivie de ses opérandes. Les objets servent ici d'opérandes. Cette catégorie d'interaction se rencontre généralement lorsque les commandes s'appliquent à plusieurs objets en fortes relations. Par contre pour une interaction objets-verbe, le dialogue est conditionné par le ou les objets et seules les commandes autorisées sont accessibles en fonction des objets. Le dialogue est plus intuitif.

Pour réaliser ces interactions, le choix du style d'interaction à un profond effet sur la nature du dialogue à engager pour accomplir son but. Il existe de nombreux styles d'interaction et notamment [SHN 91] :

- les menus;
- les langages de commandes;
- le langage naturel;

- les questions et les réponses;
- les formulaires;
- la manipulation directe.

3.1. Les menus.

Un système est guidé par les menus si chaque réponse de l'utilisateur est prise dans un ensemble de choix fourni par le système. Le système présente à l'utilisateur un ensemble de menus, chaque menu contenant une liste d'items [ART 87; KUR 93; KUR 94]. L'utilisateur contrôle le système uniquement à travers la sélection d'un item textuel ou graphique qui entraîne l'exécution d'une action. Par la sélection d'une séquence d'items, l'utilisateur traverse un réseau de menus qui peut devenir très important et très complexe.

Dans une telle approche, il est sous entendu que seuls les items correspondant à des options valides sont présentés à l'utilisateur ce qui évite un grand nombre d'erreurs de manipulation [KAN 89]. De cette manière, l'utilisateur n'est pas sollicité en termes de mémoire mais plutôt en termes de reconnaissance : il n'a pas à se souvenir de l'item qu'il désire mais à le reconnaître. Lorsque l'on parle de mémoire, l'utilisateur doit se rappeler d'une commande ou d'un concept pour entrer la donnée dans le système alors qu'en reconnaissance, les items sont associés à des connaissance déjà familières [FOL 90]. L'utilisateur se trouve alors moins sollicité et mieux guidé par des aides appropriées pour chaque menu réduisant le recours à un manuel.

Le plus souvent les menus sont utilisés de manière hiérarchique [WOO 88]. La difficulté majeure de cette représentation est de décider quels items incorporer aux différents niveaux et quels items grouper par catégorie. L'objectif est une hiérarchie homogène qui minimise le temps moyen d'accès aux fonctions, temps qui augmente avec l'accroissement du nombre d'actions et de la hiérarchie de menus [FIS 90]. Pour cela, les items doivent être groupés de manière logique pour aider la reconnaissance car très souvent l'item requis n'est pas disponible au sommet de la hiérarchie. Ainsi, les menus indiquent la structure fonctionnelle d'un système, ils reflètent à la fois l'ensemble des actions accessibles et les relations entres ses actions grâce à la structure hiérarchique [SHO 90].

Ce style d'interaction est très attractif pour un utilisateur novice. Il peut explorer les opérations fournies par le nouveau système, simplement en parcourant les menus, et devenir productif avec un nouveau système après un temps très court. Aucune syntaxe particulière n'est à apprendre et les menus s'adaptent très bien à toutes les catégories d'interaction. Des menus globaux répondent aux interactions de type verbe-objets alors que des menus contextuels (menus qui dépendent par exemple de l'objet sélectionné) se destinent aux interactions objets-verbe.

3.2. Les langages de commandes.

Ces langages sont issus directement des techniques de compilation qui sont maintenant bien connues. L'utilisateur exprime simplement les instructions au système par l'intermédiaire de touches de fonction,

de caractères simples ou d'abbréviations. Le plus souvent, les commandes sont données dans un certain langage.

La difficulté majeure pour un utilisateur réside dans la connaissance de ce langage de commandes : s'il ne le connaît pas, aucun dialogue n'est possible. L'accès à une commande se fait directement; dès lors que l'utilisateur connaît son nom, il n'est pas nécessaire de parcourir une importante hiérarchie de menus pour l'atteindre. Ce mode de fonctionnement impose à l'utilisateur de se souvenir de l'ensemble des entrées légales et de l'état du système [MOR 93]. Par opposition aux menus, l'utilisateur n'est pas guidé et les erreurs sont fréquentes car il est possible d'entrer des commandes invalides ou erronées. Dans cette situation, il ne faut pas perdre de vue que l'utilisateur n'est pas un programmeur et lui fournir des aides appropriées en cas d'erreurs.

Le temps nécessaire à son apprentissage rend un langage de commandes plus particulièrement destiné aux utilisateurs expérimentés. Pour en faciliter l'accès, il est toutefois possible d'utiliser des termes appropriés et ayant un sens pour l'utilisateur; on parle alors de langages métiers. Mais, malheureusement, le plus souvent les commandes sont peu explicites et varient suivant les systèmes ce qui ne facilite pas cet apprentissage [FOL 90].

Les langages de commandes sont fondamentalement orientés verbe-objets, l'utilisateur choisit la commande puis les objets sur lesquels elle porte. Chaque commande est paramétrée pour s'adapter aux données à traiter et peut être appliquée à de multiples objets, ce qui rend ce style bien adapté aux tâches répétitives. De plus, de par sa représentation même (langage), il est facile à étendre (nouveaux termes du langage) sans modifier sa structure. Des travaux ont également montré que l'utilisation de règles syntaxiques et sémantiques permettent de tester la consistance et la validité de tels langages [FIR 91].

3.3. Le langage naturel.

Le langage naturel est certainement la façon qui paraît la plus attractive au moins en première approche. Cependant, de nombreuses difficultés sont à surmonter. Par exemple, la formulation même d'une requête pose de nombreux problèmes concernant la reconnaissance de la voix [PIE 87]. Et si l'expression écrite élimine le problème de l'intonation et de l'accent, elle ajoute notamment les erreurs de frappe et les variations dans l'épellation des mots. Toutes ces difficultés, rencontrées dès la phase initiale du dialogue, ne favorisent pas l'analyse grammaticale et donc la compréhension des requêtes introduites par l'utilisateur.

Une fois la requête formulée, le système doit se satisfaire des constructions souvent vagues et ambiguës que l'on rencontre fréquemment. La signification d'un terme varie selon son contexte d'utilisation; nous levons tous les jours ces ambiguïtés mais il est difficile à l'heure actuelle de fournir cette capacité à la machine. Par exemple, la structure même d'une phrase peut ne pas être claire [DIX 93]. Dans la phrase, "l'homme frappe le garçon avec le bâton", le bâton appartient-il au garçon ou est-ce avec le bâton que l'homme frappe le garçon ? Trop souvent ces ambiguïtés entraînent la formulation de requêtes qui ne peuvent être satisfaites.

Des interfaces à langage naturel sont déjà disponibles mais elles se limitent à des domaines restreints et à des cas très particuliers (médecine par exemple). Non seulement le vocabulaire est limité mais la grammaire l'est également. Dans ce cas, ces systèmes ne peuvent prétendre utiliser un langage naturel alors que l'utilisateur doit apprendre des phrases. De plus, ils sont loin de la flexibilité et de la facilité de communication escomptées. Certains se posent même la question de l'utilité même du langage naturel qui est, par définition, vague et imprécis (flexible et créatif) alors que les ordinateurs ont besoin d'instructions précises.

La difficulté de ce style d'interaction provient probablement de ce que l'on attend de lui. Il rend de très grands services dans des domaines de pointe sans pour autant être destiné à n'importe quel utilisateur (en médecine, un chirurgien donne des ordres à un appareil sans avoir à lâcher ses instruments). Mais dans ce cas, on se rapproche plus d'un système à base de menus ou d'un langage de commandes auquel on associe une interface plus "humaine" et le plus souvent orale.

3.4. Les formulaires.

Les formulaires sont utilisés pour l'entrée de données ou pour les affichages des applications de recherche d'informations (par exemple dans les sytèmes d'informations). Ils se présentent sous la forme de boîtes comportant des champs et un aspect que l'utilisateur à l'habitude de voir.

De cette manière, l'utilisateur travaille avec le formulaire et l'application reçoit les données en bonne position et avec des valeurs correctes. Pour cela, les formulaires doivent être définis de manière à permettre à l'utilisateur de clairement distinguer les sortes de données demandées. Ainsi, en utilisant des commandes pour passer d'un champ à un autre et corriger un champ, l'utilisateur n'a pas à se préoccuper de se positionner au bon endroit et n'a pas besoin de regarder l'écran avec trop d'attention.

Ce style d'interaction convient très bien lorsque plusieurs types de données sont demandés par le système; dans ce cas, il est aisé de considérer l'écran comme un formulaire. Non seulement un tel style est facile à utiliser et à apprendre mais il autorise un certain nombre d'anticipations puisque tous les champs à remplir sont visibles. L'utilisateur sait à tout instant où il va [FOL 90].

Les grilles représentent une variante des formulaires. Une grille est composée d'un ensemble de cellules pouvant contenir une valeur ou une formule. Chaque formule peut faire référence à d'autres cellules pour effectuer certains calculs (calculs financiers, calculs de contraintes [MYE 91], ...). L'utilisateur peut changer n'importe quelle cellule et le système maintient la consistance des valeurs affichées assurant que la formule est satisfaite. Les grilles se placent au centre de l'interaction : l'utilisateur est libre de manipuler des valeurs à volonté, il n'y a pas de restriction, et la frontière entre entrée et sortie est floue. Elles sont très utilisées, notamment dans les feuilles de calculs des tableurs, car leur utilisation est naturelle et flexible; l'utilisateur peut essayer plusieurs alternatives et voir instantanément le résultat sur les cellules de la grille. On rejoint alors le processus essai-erreur que l'on trouve dans les modèles d'interaction.

3.5. Les questions et réponses.

Ce style fait référence à un type rudimentaire de systèmes où chaque interaction (questions, ensembles de choix et réponses de l'utilisateur) se limitait à une ou deux lignes d'affichage [PRE 94]. Le système pose une série de questions à l'utilisateur (réponse oui-non, choix multiples, ...), chaque question dépendant des réponses précédentes. La réponse de l'utilisateur est contrainte à un ensemble de réponses attendues et au cas où cet ensemble est petit, la question peut inclure les réponses possibles .

Il s'agit d'un simple mécanisme pour fournir des informations à une application d'un domaine spécifique. Le dialogue est initié par l'application. En effet, le contrôle est interne, l'utilisateur n'a pas à se préoccuper de navigation. Cependant, cela pose un problème de contexte car l'utilisateur a uniquement le contexte des questions précédentes et courantes pour se repérer et influencer ses réponses. Il ne peut anticiper les questions contrairement aux formulaires où il voit tous les champs à remplir.

Ce style d'interaction est bien adapté à des domaines restreints (certains systèmes d'information) ou lorsque l'information fournie par l'utilisateur est sous forme limitée et ordonnée. Il est facile pour les utilisateurs débutants (l'application contrôle les dialogues et l'utilisateur est guidé), mais limité en fonctionnalité et en puissance notamment pour un expert qui sait ce qu'il veut faire.

3.6. La manipulation directe.

Le concept de manipulation directe a été introduit en 1983 [SHN 83]. L'idée de départ est d'éviter l'apprentissage d'une syntaxe même la plus simple par la manipulation directe de l'objet de l'intérêt et de ne requérir ainsi qu'un simple entraînement. Pour cela, ce concept autorise l'utilisateur à exécuter une commande à tout moment par opposition aux dialogues modaux qui ne permettent que certaines commandes dans un certain mode.

Une interface à manipulation directe propose un ensemble de représentations visuelles (textuelles ou graphiques) sur un écran et un répertoire de manipulations qui peuvent s'y appliquer [JAC 86]. Les commandes ne sont pas appelées explicitement par des moyens traditionnels; elles sont implicites dans l'action que réalise l'utilisateur sur la représentation visuelle. L'utilisateur a alors l'impression de travailler directement avec l'objet au lieu de manipuler un dialogue agissant sur lui.

Pour cela, une interface à manipulation directe doit posséder les caractéristiques suivantes [PAN 93] :

- l'affichage doit être le reflet de l'état courant du système et en particulier les objets manipulés par l'utilisateur doivent être continuellement visibles;
- les opérations doivent être rapides, incrémentales et réversibles avec un impact rapidement visible;
- le langage de commandes doit être remplacé par la manipulation directe de l'objet de l'intérêt.

Bien entendu, le développement de telles interfaces fait apparaître des difficultés supplémentaires de

part les manipulations des objets et les réactions très rapides rendues nécessaires pour garantir une bonne interactivité [MYE 89].

Ce style est à la base des interactions objet-verbe : l'utilisateur désigne en tout premier lieu le ou les objets puis la commande à appliquer. Par ce biais, la distance cognitive (distance la plus courte existant entre l'objet et sa manipulation) et la distance d'erreur (distance entre la manière de penser d'un utilisateur et la façon avec laquelle le système la représente) sont réduites [FRO 93]. La manipulation directe diminue considérablement la réflexion et l'adaptation nécessaire de l'utilisateur. Elle est très appréciée car elle est naturelle, rapide et même amusante. Un utilisateur néophyte peut apprendre rapidement les fonctions de base car il voit immédiatement si ses actions vont dans le sens qu'il désire; dans le cas contraire il peut simplement changer de direction. Même la majorité des concepteurs préfèrent les outils à manipulation directe car ils n'apprécient guère les outils complexes de développement qui requièrent l'apprentissage de langages spécifiques aux règles compliquées [CAI 92].

Cependant, ce style d'interaction est trop souvent présenté comme le meilleur; il est facile à utiliser mais il manque parfois de puissance [FOL 90]. En effet, il peut être lent pour un utilisateur expérimenté alors qu'un autre style serait plus approprié. De plus, toutes les tâches ne peuvent être réalisées avec des objets concrets (tampon lors de l'opération de copie, ...) et la manipulation directe présente des limitations concernant l'abstraction. C'est pourquoi, très souvent, ces interfaces sont couplées avec le concept de programmation par l'exemple qui permet après la manipulation d'un exemple concret de le généraliser à un ensemble d'objets [MYE 92a; MYE 93].

3.7. Conclusion.

Dans les paragraphes précédents, la plupart des styles d'interaction ont été décrits séparément. Une constatation s'impose : il n'existe pas à l'heure actuelle un style d'interaction universel. Il ne faut pas considérer ces styles comme exclusifs, la solution n'étant certainement pas dans l'utilisation d'un style mais dans une combinaison. À chaque interaction, il faut faire correspondre le style adapté en fonction du contexte et de l'utilisateur. Par exemple, pour un utilisateur néophyte on préférera un style rapide à assimiler comme les menus (graphiques de préférence) ou la manipulation directe.

Beaucoup de facteurs peuvent affecter la qualité de l'interaction, que ce soient des facteurs sociaux, physiques, ... Il est nécessaire d'utiliser des métaphores en fonction de ces facteurs (pour développer une image du système qui correspond au modèle de l'utilisateur) et de motiver l'utilisateur en lui fournissant des échos adéquats de son travail (pour s'assurer que l'interaction donne le résultat requis). Dans le cas contraire, il pourrait même croire que ses actions n'ont pas été réalisées avec succès. L'important est de fournir des outils pour prévenir cette confusion et autoriser la correction des erreurs [DIX 93].

Mais, au delà de l'utilisateur lui-même, on doit tenir compte de l'application. Un style langage de commandes ne peut être utilisé dans le cadre d'un dialogue objets-verbe alors que les menus s'adaptent

très bien dans tous les cas. Ainsi, beaucoup de systèmes s'appuient sur une combinaison de ces styles. On peut citer par exemple l'utilisation d'un langage de commandes et de menus dans [KAN 89].

4. LES MODÈLES D'ARCHITECTURE.

Un modèle d'architecture se fonde sur l'analyse descendante d'un problème. C'est un modèle abstrait qui doit fournir au programmeur d'interfaces une structure générique pour faciliter la conception et le développement des systèmes interactifs en identifiant les différents composants susceptibles d'être développés avec une relative indépendance [CHA 93; PAN 93]. Il doit également contenir la description des échanges de données entre l'utilisateur et l'application, les étapes de transformation des données ainsi que l'agencement des composants qui assurent ces transformations [COU 90; DUV 91].

Tous les modèles proposés ont comme objectif initial de séparer le plus convenablement possible ce qui est application proprement dite de ce qui est dialogue avec l'utilisateur. On peut distinguer deux grandes familles :

- les modèles globaux;
- les modèles éclatés ou multi-agents.

Les modèles globaux sont des modèles conceptuels qui décrivent un système à travers ses composants essentiels. Ils fournissent un cadre méthodologique pour séparer les difficultés en plusieurs modules. Ils n'abordent ni l'architecture interne (ils ne tiennent pas compte des objets), ni les interfaces d'échange.

Les modèle éclatés que l'on peut rapprocher de la conception objets représentent le système à un niveau plus faible. La séparation en composants ne se fait pas pour le système dans sa globalité. Il est constitué d'un ensemble d'objets ou d'agents qui regroupent chacun des morceaux de l'application et du dialogue. On obtient une décomposition quasi récursive d'un système et une granularité plus fine.

Nous présentons trois modèles très connus. La présentation du modèle global Seeheim et du modèle éclaté PAC est l'occasion d'aborder le problème de l'interactivité et de la circulation des données à travers l'étude du modèle IDS.

4.1. Le modèle Seeheim.

Ce modèle très général et très connu ne dépend pas directement d'un système implanté [GRE 86]. Il a été développé avant toute implantation et doit son nom au congrès sur les systèmes de gestion d'interfaces utilisateur qui s'est déroulé à Seeheim, ex-RDA, en 1983 [DUV 93; DUV 94]. Depuis, de nombreux systèmes l'ont utilisé de manière plus ou moins stricte.

Sa définition s'inspire du dialogue entre individus. Comme le modèle linguistique, il se décompose en plusieurs composants : *Présentation*, *Contrôle du dialogue* et *Interface avec l'application*. Il est à noter que le schéma initial ne comporte ni l'utilisateur, ni l'application; cela suppose que les concepteurs de

ce modèle se sont basés sur un contrôle externe du dialogue sinon ils auraient fait apparaître l'application.

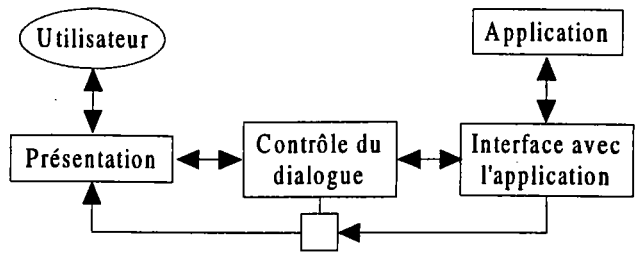


Figure I.1. *Le modèle SEEHEIM.*

Le composant *Présentation* représente le niveau lexical de l'interface. Il est responsable de la représentation externe du système et de son interface utilisateur; il définit l'image du système auprès de l'utilisateur. C'est le seul composant à être directement en relation avec les périphériques d'entrées/sorties. Les autres composants ne peuvent échanger directement des informations avec les périphériques. *Présentation* convertit les informations entre le monde physique de l'utilisateur et le monde informatique. Il est responsable de la lecture des données en provenance des dispositifs physiques d'entrées (termes du langage d'interaction), des styles d'interaction mais aussi des sorties élémentaires telles que les lignes, les boîtes, ...

Le composant *Contrôle du dialogue* est le niveau syntaxique de l'interface. C'est le médiateur entre les deux autres composants. Il est chargé du dialogue entre l'utilisateur et le système avec, entre autres, la gestion de l'état de l'interaction, des états possibles et la définition des phrases définissant la structure du dialogue [SAD 92]. Ainsi, il effectue une analyse syntaxique des termes reçus du composant *Présentation* afin de construire des phrases correctes et les transmettre à l'application. Inversement, il ventile les phrases provenant de l'application vers le composant *Présentation*. Bien entendu, il peut effectuer certaines vérifications sémantiques avant d'envoyer une requête au composant *Interface avec l'application.*

Le composant *Interface avec l'application* est une passerelle entre l'interface utilisateur et le reste du programme, il gère les appels aux modules de l'application. C'est le représentant de l'application auprès du composant *Contrôle du dialogue*. Il représente la sémantique du langage d'interaction grâce aux concepts et aux actions qu'il fournit (description des structures de données et des modules accessibles). Des mécanismes de conversion doivent être fournis entre les concepts de l'application et les phrases du composant *Contrôle du dialogue* (par exemple, un nom suivi de paramètres). Ce composant peut également contenir une description des contraintes d'utilisation pour vérifier la validité des requêtes de l'utilisateur avant d'appeler les services de l'application.

Ce modèle hérite des avantages et inconvénients des modèles linguistiques qui sont bien connus dans leur principe; ses trois composants peuvent être vus comme des processus indépendants qui communiquent à l'aide de symboles similaires à ceux utilisés en compilation [AHO 90]. Les symboles d'entrée sont représentés par les données de l'utilisateur et les symboles de sortie par les informations à

fournir à l'utilisateur. Par contre, ses trois niveaux entraînent une certaine lourdeur pour les échos sémantiques et s'adapte difficilement aux dialogues non modaux. De plus, cette structure monolithique entraîne que toute modification sur un composant se répercute sur l'autre composant en relation. Chaque composant étant de taille respectable pour une application donnée, il est aisé d'imaginer les difficultés de maintenance.

Ce modèle est plus une synthèse de ce qui se faisait, qu'une réelle évolution [DIX 93]. En effet, il n'aborde pas les problèmes actuels des applications graphiques à forte sémantique ou de la réutilisation pour la contruction d'importants systèmes interactifs. Des variantes de ce modèle existent et elles s'intéressent principalement à des découpages différents des composants. On peut citer par exemple le modèle Seeheim étendu, le modèle ARCHE ... Par contre, le modèle IDS tente d'apporter des réponses au problème des interfaces à forte sémantique.

4.2. Le modèle IDS.

Concernant la sémantique, l'ensemble des traitements de reconnaissance des phrases sont rassemblés en un seul bloc, alors que certaines opérations lexicales de base nécessitent une connaissance sémantique pour être traitées. Les modèles de type Seeheim s'appuient sur la séparation entre la sémantique et la présentation, séparation qui apparaît très stricte. En effet, une séparation aussi nette entraîne une communication dense; l'interactivité requise dans les interfaces de haut niveau devient alors coûteuse voire impossible à réaliser [MYE 89]. De plus, il est difficile de construire des outils génériques de développement d'interface en s'appuyant sur ces architectures. Le composant *Contrôle du dialogue* est très dépendant des deux autres composants, ce qui entraîne sa réécriture en tout ou partie dès que les autres composants sont modifiés.

Ainsi, des améliorations successives ont été apportées au modèle Seeheim afin de fournir une meilleure répartition des données : Seeheim modifié, Seeheim étendu. Une description plus complète des différents modèles dérivés est donnée dans [DUV 94]. Le problème de la répartition des données consiste à rechercher un équilibre entre les informations internes au gestionnaire de dialogue et internes à l'application. Dans une première analyse, [DAN 87] étudie les besoins en informations sémantiques (S) à la fois pour le dialogue (D) et pour l'application (A). En pratique, l'approche (D S) (A) où la sémantique se trouve dans le dialogue est plus adaptée pour les interfaces qui doivent supporter les mises à jour d'affichage ou la visualisation des données spécifiques. Le dialogue peut contenir des routines standard. L'extension d'une telle approche est toutefois difficile car le système s'appuie sur les données existantes dans le dialogue. Par contre, l'approche (D) (S A) où la sémantique se trouve dans l'application est plus souple mais elle nécessite un travail plus important car tous les traitements de mise à jour et d'affichage sont à développer. Ces deux approches peuvent être complémentaires, le modèle IDS (Interface par Délégation Sémantique) en est un exemple [CHA 93].

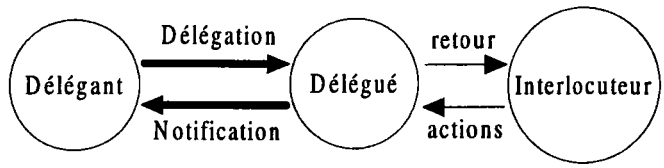


Figure I.2. *Le modèle IDS.*

Dans ce modèle, l'interlocuteur interagit avec le délégué qui contient la sémantique qui lui a été fournie (déléguée) par le délégant. Une action peut donc être réalisée sans recours au délégant même si une forte sémantique est nécessaire. Lorsque l'action est réalisée, le délégué notifie le délégant de l'opération qui vient de se réaliser, celui-ci pouvant tout de même refuser l'opération.

La sémantique se répartit (entre le Délégant et le Délégué) selon les besoins toujours dans le but de respecter la dynamicité (diminution des échanges) et l'indépendance (autonomie du dialogue). Toutefois, ce modèle ne mesure pas le coût de la délégation de cette sémantique. En fonction du contexte du dialogue, la sémantique déléguée doit souvent être remise à jour. De plus, aucune méthode n'est fournie pour déterminer quelles informations doivent être déléguées pour assurer une bonne dynamicité tout en conservant une partie de cette sémantique pour ne pas surcharger le dialogue. Le principe de ce modèle est intéressant mais la difficulté de son évaluation et sa très forte dépendance par rapport aux données manipulées orientent son utilisation vers des domaines très ciblés, clairement définis ou pour des études abstraites d'échange de l'information.

4.3. Un modèle multi-agents : PAC.

Les modèles multi-agents tentent d'apporter une réponse au problème de la réutilisation en étendant le découpage à l'ensemble des objets de l'application. Ils décomposent le système en agents spécialisés et répartissent le dialogue sur l'ensemble de ces agents. Un agent s'inspire du fonctionnement par stimuli-réponses. Il est constitué d'un ensemble de récepteurs, d'émetteurs et de variables d'état. Lorsqu'un émetteur produit un événement, les récepteurs sensibles à cet événement sont activés et l'événement est traité par l'agent propriétaire de ces récepteurs. Le traitement peut être un changement d'état, l'émission de nouveaux événements, l'exécution d'actions, ... Ainsi, le modèle multi-agent structure un système interactif en un ensemble d'agents réagissant à des événements et produisant des événements. L'organisation décrite fait apparaître une grande modularité et des traitements distribués, caractéristiques communes avec les modèles objets [CHA 93; SAD 92].

Le modèle PAC est un modèle multi-agents qui s'inspire du modèle Seeheim [COU 90]. Il est basé sur le flou existant entre la notion d'application et la notion d'interface. Il conserve la décomposition des traitements au niveau lexical, syntaxique et sémantique mais avec une approche orientée objets. La syntaxe et la sémantique ne forment pas deux blocs monolithiques mais plusieurs composants à des niveaux d'abstraction différents. C'est la première interprétation modulaire ou répartie de Seeheim. Une application est décomposée en un ensemble hiérarchique d'objets PAC, la connexion entre chaque niveau fonctionnel s'effectuant à l'intérieur des objets. Chaque objet comporte trois facettes (ou composants) qui implantent les trois composants de Seeheim.

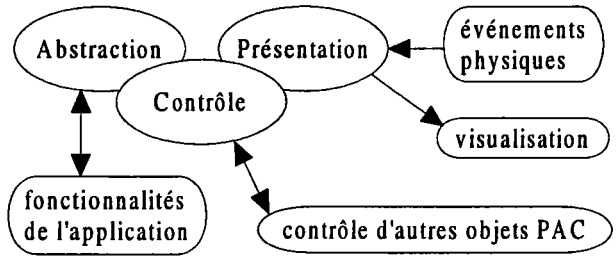


Figure I.3. *Un objet PAC.*

La facette *Présentation* définit l'image de l'objet pour l'utilisateur ainsi que son comportement non seulement en entrées mais aussi en sorties.

La facette *Contrôle* est chargée du lien et du maintient de la cohérence entre les facettes *Présentation* et *Abstraction.*

La facette *Abstraction* représente la sémantique de l'objet, à savoir des fonctions ou des attributs liés à la fonction de l'objet. C'est la partie visible par les autres modules du système.

Le modèle PAC peut être comparé à un modèle objets à structure homogène; tous les objets ont la même structure, ils contiennent leur représentation, leur contrôle et leur sémantique. Cette séparation entre les facettes n'est que logique alors que la séparation physique entre les objets facilite la conception, la modularité et le parallélisme. Cependant, il n'est pas toujours facile de concilier ce modèle avec les impératifs de la programmation. D'une part, le réseau des communications entre les objets PAC peut devenir très complexe et d'autre part la structure logique d'un tel environnement est très compliquée comparée à une interface traditionnelle [FAC 92]. Même si certains systèmes comme [BRU 93] l'utilisent, un tel modèle s'apparente à une architecture conceptuelle car il ne tient pas compte des problèmes et des contraintes d'implémentation [DIX 93; DUV 91].

4.4. Conclusion.

Le dénominateur commun de ces modèles d'architecture est la recherche de la séparation entre le dialogue et l'application. On peut constater que cette séparation se fait au prix d'une imprécision générale. Aucun modèle ne définit clairement les interfaces mises en place entre les différents composants, composants qui ne sont même pas spécifiés entièrement. Une réponse serait de définir le domaine d'application visé par ces modèles. En effet, un modèle d'architecture ne peut que devenir dépendant des difficultés qu'il doit résoudre. On l'a vu pour la délégation et le modèle IDS, sans information sur les données, on ne peut fournir de méthodes pour déterminer les données déléguées et donc les échanges mis en place.

Les modèles globaux possèdent une structure monolithique qui s'adapte mal aux nouveaux concepts de Programmation Orientée Objets (POO). Ils présentent un atout non négligeable pour les systèmes qui manipulent des objets possédant de fortes relations. La portabilité de tels systèmes est facilitée par la séparation stricte et bien définie entre le dialogue et l'application.

Les modèles éclatés sont plus adaptés aux concepts de la POO avec laquelle la parenté est évidente. Cependant, ils ne définissent pas (comme les langages orientés objets d'ailleurs) les critères à utiliser pour identifier et structurer les agents. Une fois les agents identifiés, il apparaît que ces modèles conviennent bien aux systèmes dont les objets possèdent peu de relations. Dans ce cas, un objet est représenté par un agent et le contrôle désigne les actions supportées par cet objet. Par contre, le modèle ne convient pas lorsque les actions s'appuient sur plusieurs objets ou si la structure des agents ne suit pas celle des objets.

Cette brève présentation fait apparaître qu'il n'existe pas de modèle d'architecture adapté à toutes les applications. Il convient de considérer plutôt des modèles par domaine pour caractériser précisément les besoins et donc les données. De même, le choix entre un modèle global ou éclaté est un faux problème, on s'oriente de plus en plus vers des modèles hybrides comme par exemple le modèle PAC-Seeheim [DUV 94].

5. LES MODÈLES DE DIALOGUE.

Devant l'augmentation constante de la complexité des interfaces homme-machine modernes, il s'est avéré nécessaire de fournir au programmeur d'interfaces un modèle servant de guide et de cadre de travail. Le modèle de dialogue est un modèle abstrait qui est utilisé pour décrire la structure du dialogue mis en place entre un utilisateur et le système [GRE 86; MYE 89]. Contrairement aux modèles d'architecture qui fournissent une vue statique du système (modélisation des différents composants en insistant sur la séparation), les modèles de dialogue se concentrent sur le comportement du système par rapport aux entrées de l'utilisateur. Ils décrivent l'aspect dynamique du système. Dans le cas du modèle Seeheim, le modèle de dialogue décrit le composant *Contrôle du dialogue*. Un tel modèle autorise la description et l'étude d'une grande variété de dialogues sans avoir à les coder [JAC 86].

Parmi ces modèles, on distingue essentiellement deux familles : les modèles de conception et les modèles d'implémentation. Les modèles de conception ont pour objectif premier de "capturer" la pensée du concepteur et s'expriment généralement sous une forme très abstraite alors que les modèles d'implémentation sont utilisés lors de l'implémentation de l'interface et sont donc très formels. Bien entendu, ces deux familles de modèles ne sont pas exclusives et doivent être considérées comme complémentaires. L'objectif de nos travaux étant de fournir une méthode d'implémentation pour les systèmes de C.F.A.O., nous nous intéresserons exclusivement aux modèles d'implémentation existant.

Au cœur du modèle de dialogue se trouve la notation choisie. À travers cette notation et l'aide apportée à la description du dialogue, les modèles de dialogue influencent fortement l'implémentation. Ainsi, par le choix de la notation, il est possible de déterminer l'ensemble des interfaces que le concepteur va pouvoir décrire. Pour produire le plus grand nombre d'interfaces, il est indispensable de disposer de la notation la plus puissante. On distingue la puissance d'utilisation, l'ensemble des interfaces que l'on peut décrire facilement, et la puissance de description, l'ensemble des interfaces que l'on peut décrire sans tenir compte de la difficulté.

Les premiers travaux sont issus de la théorie des langages et de la compilation (automates). On trouve en particulier les modèles basés sur les grammaires [BAR 89] et sur les réseaux de transitions [BOR 92; CAI 92; CHU 88; DAN 87; JAC 86; WAS 85]. Après ces modèles que l'on peut qualifier de globaux à quelques exceptions près (le dialogue est décrit comme une et une seule entité), des modèles plus répartis sont apparus et notamment le modèle à événements [FOL 90]. Actuellement des modèles très différents sont étudiés comme par exemple les langages déclaratifs [MYE 92b]. Nous étudions ces modèles dans les paragraphes suivants pour évaluer leur puissance et leur diversité.

5.1. Les grammaires.

La théorie des langages, qui est bien maîtrisée grâce à la compilation, a constitué le point de départ de la modélisation de la dynamique des dialogues. L'idée de décrire les dialogues entre l'homme et la machine par des grammaires n'est pas surprenante. Ce modèle considère l'interaction homme-machine comme un dialogue, au même titre que la communication homme-homme [DIX 93; HAR 90]. Dans le cas du langage naturel, on utilise une grammaire afin de représenter le langage des participants à la discussion. L'extension naturelle de cette idée est d'utiliser une grammaire pour spécifier le dialogue homme-machine. La différence essentielle est que la communication entre deux personnes passe généralement par un seul langage alors que la communication entre l'homme et la machine en impose deux. Nous nous intéressons exclusivement au langage de l'homme vers la machine.

5.1.1. Les grammaires hors contexte.

Elles sont utilisées pour définir précisément les séquences autorisées des mots du langage à partir d'une représentation textuelle compréhensible par un informaticien. On parle de grammaires d'actions et de réponses [GRE 86]. Une grammaire hors contexte permet la description de la syntaxe d'un langage à partir d'un ensemble de lexèmes appelés terminaux. Ces terminaux sont combinés par les règles de production pour former des structures de plus haut niveau que l'on appelle des non-terminaux. L'ensemble des règles de production ainsi décrit forme la grammaire du langage et sert alors à vérifier la validité du discours.

Pour la définition du dialogue homme-machine et en s'appuyant sur le modèle d'architecture Seeheim, on peut définir les terminaux par les objets (ou codes) générés par le composant *Présentation* en réponse aux actions de l'opérateur. De cette manière, l'ensemble des règles de production forme les interactions que l'utilisateur peut avoir avec le système (la syntaxe d'interaction). Une grammaire de construction d'un rectangle pourrait être :

RECTANGLE	←	ORIGINE COIN
ORIGINE	←	bouton
COIN	←	déplacement COIN
	|	bouton

Dans cet exemple, les non-terminaux sont représentés en majuscules alors que les terminaux sont en

minuscules. L'opérateur | autorise plusieurs règles de construction pour une règle de production. Le terminal *bouton* représente la désignation d'un couple de coordonnées par l'utilisateur alors que *déplacement* représente le déplacement du pointeur de désignation. La règle RECTANGLE indique que le cadre est défini par une origine et un coin. La règle ORIGINE fixe l'obtention de l'origine par une désignation (terminal *bouton*) alors que la règle COIN indique que l'extrémité est également obtenue par une désignation, mais après d'éventuels déplacements. Cette dernière règle montre que la définition récursive des non-terminaux autorise la représentation des itérations.

L'objectif est de modéliser les actions de l'utilisateur et les réponses du système. Cependant, la grammaire présentée précédemment décrit seulement les actions possibles de l'utilisateur mais ne spécifie pas les réponses que peut donner le système à ces actions. Pour cela, il est possible d'attacher des actions aux productions. Par exemple, au cours du dialogue de création du rectangle nous devons enregistrer les points pour construire ensuite le rectangle. On obtient la grammaire suivante :

CADRE	←	ORIGINE	COIN	
ORIGINE	←	bouton	ORIGINE_OK	
COIN	←	déplacement	DESSINER	COIN
	|	bouton	COIN_OK	
ORIGINE_OK	←	{ enregistrer l'origine }		
DESSINER	←	{ dessiner le cadre }		
COIN_OK	←	{ enregistrer le coin }		

Les grammaires donnent une structure naturelle à la description des dialogues séquentiels par une concentration sur l'action. Elles sont bien connues grâce aux techniques de compilation et à des générateurs tels que YACC (notamment les grammaires de type LR et LALR) [AHO 90]. Même s'il est possible, dans certains cas, d'évaluer certains aspects concernant la qualité du dialogue généré [FOL 90], les grammaires hors contexte posent un certain nombre de difficultés. L'ensemble des règles forme un tout indissociable dont la compréhension est rendue difficile lorsqu'il augmente. De plus, ce caractère mono-bloc ne permet pas la représentation des fils d'activité multiples, d'autant que l'on ne peut pas décrire des interfaces dont la réponse dépend de l'état du système. En effet, il est impossible de représenter la réaction du système dans la mesure où l'application ne peut pas influencer le contrôleur de dialogue représenté par le générateur [GRE 86; MYE 89]. En particulier, cette limitation ne favorise pas la description d'aides contextuelles et plus généralement, on peut reprocher un manque de support des erreurs, pourtant très demandé dans les interfaces actuelles. Seules les phrases licites sont considérées et le comportement opportuniste de l'utilisateur ne peut être pris en charge. De nombreuses extensions ont été étudiées pour résoudre les limitations des grammaires et principalement celles liées aux interfaces contextuelles et aux dialogues concurrents. Nous étudions par la suite les règles de production et les algèbres de processus.

5.1.2. Les règles de production.

Dans ce modèle, une grammaire est constituée de règles s'exprimant sous la forme d'une condition associée à une action : CONDITION ➔ ACTION. Ces règles peuvent être orientées événement ou orientées état.

Dans le premier cas, les conditions et les actions sont représentées par un ensemble d'événements. Parmi ces événements, on distingue trois sortes : les événements utilisateur issus des actions de l'utilisateur, les événements internes définissant l'état du dialogue et les événements réponses décrivant les actions du système. Quant au système, il est représenté par un ensemble d'événements : SYSTÈME = { événements }. On dit qu'une règle est déclenchée si l'ensemble des événements de la condition sont présents dans le système. Les événements de la condition ayant servi au déclenchement sont alors retirés du système et les événements décrits dans l'action sont ajoutés. Comme exemple, les trois règles suivantes autorisent la sélection d'une option pour la construction d'un rectangle (les événements de l'utilisateur sont en majuscule, les événements internes en minuscules et les événements réponses entre '<' et '>') :

SÉLECTION_RECTANGLE		➔	début_rectangle
début_rectangle	POINT	➔	fin_rectangle
fin_rectangle	POINT	➔	< dessiner le rectangle >

Dans le cas des règles orientées état, le système est composé d'un ensemble d'attributs, chaque attribut possédant un certain nombre de valeurs. La valeur d'un attribut est conservée tant qu'un changement explicite n'est pas effectué contrairement au modèle précédent où les événements étaient retirés automatiquement. On parle de la persistance des attributs. Nous prenons comme exemple un dialogue où l'utilisateur peut choisir le style d'un texte, italique et/ou gras. L'ensemble des attributs est :

SOURIS	=	{	pas_de_sélection	sélection_italique	sélection_gras }
ITALIQUE	=	{	on	off	}
GRAS	=	{	on	off	}

L'ensemble des règles est :

sélection_italique	ITALIQUE=on?	➔	pas_de_sélection	< ITALIQUE = off >
sélection_italique	ITALIQUE=off?	➔	pas_de_sélection	< ITALIQUE = on >
sélection_gras	GRAS=on?	➔	pas_de_sélection	< GRAS = off >
sélection_gras	GRAS=off?	➔	pas_de_sélection	< GRAS = on >

La persistance présente des avantages non négligeables, dans la mesure où le programmeur d'interfaces ne s'intéresse qu'aux attributs dont il désire modifier la valeur, mais elle impose de modifier explicitement toutes les valeurs. Il est par exemple nécessaire de réinitialiser à chaque fois explicitement la souris, sélection_gras vers pas_de_sélection, sous peine de bouclage. Comme on le

voit, chaque règle est décrite indépendamment et l'ensemble constitue un dialogue à fils d'activités multiples (chaque règle est une possibilité de dialogue contextuel). On peut ajouter très facilement un autre style (souligné par exemple) sans affecter les règles déjà écrites. En fonction du nombre des dialogues, le nombre des règles peut bien entendu devenir très important.

Les règles de production sont bien adaptées pour les dialogues à fils d'activités multiples et même concurrents. En effet, selon ce modèle, les règles dépendent du contexte représenté par l'état du système (SYSTEME) et plusieurs règles peuvent être candidates pour un même état (par exemple les quatre règles pour le style d'écriture). À l'opposé, les dialogues séquentiels doivent être codés à l'aide d'événements internes ou de valeurs particulières (états début_rectangle et fin_rectangle de l'exemple). Ce codage alourdit beaucoup la syntaxe et ne facilite pas la compréhension d'une telle spécification.

5.1.3. Les algèbres de processus.

Les algèbres de processus sont développées pour s'adapter, au mieux, aux dialogues séquentiels et aux dialogues concurrents. Une des notations utilisée, les processus séquentiels communiquants (CSP pour Communicating Séquential Process), s'apparente aux grammaires dans son fonctionnement de base [ABO 90].

Les règles sont définies comme des processus et leur écriture est simplement facilitée par l'ajout de différents symboles et opérateurs. Ainsi, l'opérateur '=' indique une définition, '→' une séquence d'événements, ';' une séquence de processus et '[]' un choix. Pour un système de dessin d'un cercle et d'un segment, on obtient :

MENU	=	sélectionner_cercle?	→ FAIRE_CERCLE			
		[] sélectionner_segment?	→ FAIRE_SEGMENT			
FAIRE_CERCLE	=	désignation?	→ ranger_centre	→ désignation?	→ dessiner_cercle	
FAIRE_SEGMENT	=	DÉBUT_SEGMENT	; FIN_SEGMENT			
DÉBUT_SEGMENT	=	désignation?	→ ranger_origine			
FIN_SEGMENT	=	désignation?	→ dessiner_segment			

Les processus sont écrits en majuscule, les événements sont suivis par un '?' et les actions sont en minuscules. À partir de cette syntaxe possédant la même puissance de description qu'une grammaire, l'utilisateur a à sa disposition l'opérateur '||' qui autorise la composition parallèle. De cette manière, deux processus P et Q peuvent être mis en parallèle comme si deux analyseurs s'exécutaient de manière concurrente, prenant chacun en compte les événements, qui le concernent alors qu'ils ont été décrits indépendamment l'un de l'autre.

On peut remarquer que les extensions apportées aux grammaires permettent d'en augmenter sensiblement la puissance. Cependant cette puissance est obtenue au prix d'une augmentation importante de la complexité des outils de génération car de nombreuses propriétés des grammaires hors contexte sont perdues. Les grammaires que l'on obtient ne répondent toujours pas au problème de la

complexité d'utilisation et notamment la difficulté d'obtenir une vue globale du dialogue spécifié [FOL 90]. Ce point est d'autant plus pénalisant que les grammaires étendues deviennent complexes et sortent même de la théorie des langages pour se rapprocher des réseaux de transitions.

5.2. Les réseaux de transitions.

C'est un formalisme graphique, simple, comparé aux grammaires précédentes [HAR 90; WAS 85]. Basé sur les graphes, il est constitué d'un ensemble d'états représentés par des cercles et d'un ensemble de transitions représentées par des flèches reliant deux états. Pour le développement du dialogue, chaque état de ce réseau représente l'état du dialogue entre l'utilisateur et le système.

Dans la version la plus simple des réseaux de transitions (STN pour State Transition Network), les transitions sont simplement étiquetées par les actions que l'utilisateur peut effectuer. Le réseau décrit, à travers les transitions, les choix possibles de l'utilisateur à un instant donné. Ils indiquent comment le dialogue évolue d'un état vers un autre. On dit qu'une transition est validée, si l'utilisateur effectue l'action qui étiquette la transition. Le système passe d'un état A à un état B, si l'une des transitions qui va de A vers B est validée. On définit alors un chemin par une séquence de transitions qui va de l'état initial à l'un des états finals. Une séquence d'actions est acceptée si elle étiquette les transitions d'un chemin du graphe. Par exemple, le réseau de construction d'un rectangle est :

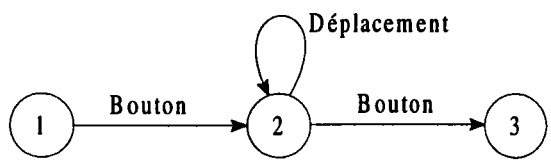


Figure I.4. *Le réseau de transitions de construction d'un rectangle.*

Cet exemple montre qu'aucune réaction du système n'est prévue (calculs, affichages, ...). Dans le cas des grammaires, les actions ont été attachées aux règles, pour les réseaux la première solution est d'attacher les actions aux états. De cette manière, lorsqu'un état est atteint son action est exécutée. Les états représentent alors les actions du système et les transitions les actions permises de l'utilisateur. Le dialogue de création du rectangle devient :

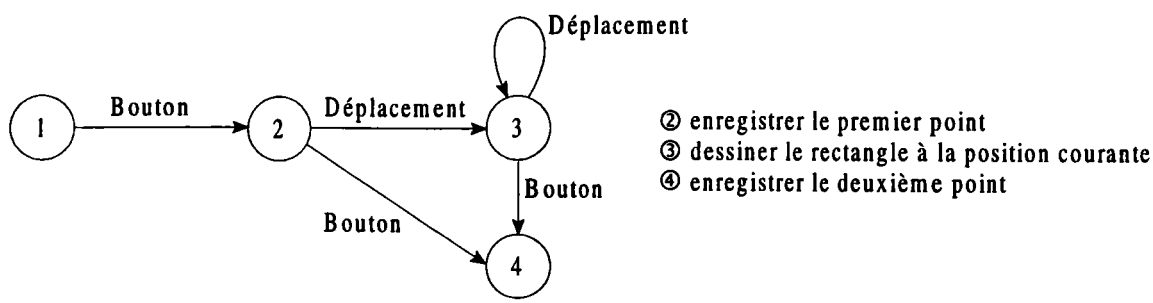


Figure I.5. *Le réseau de transitions de construction d'un rectangle (association état-action).*

La seconde solution est d'attacher les actions aux transitions. Dans certains cas, le réseau obtenu est plus simple et surtout un grand nombre d'états fantômes disparaissent (par exemple l'état 3 du réseau

précédent). Ainsi, lorsque la transition est validée, son action est exécutée.

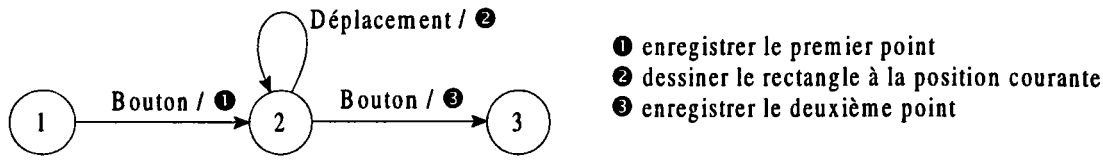


Figure I.6. *Le réseau de transitions de construction d'un rectangle (association transition-action).*

L'intérêt principal de ces réseaux est de pouvoir analyser rapidement la structure du dialogue. Ils représentent explicitement les différents états qui caractérisent la situation du dialogue et facilitent ainsi la compréhension. Cependant, l'augmentation exponentielle du nombre des états pose un problème avec des structures très importantes et des dialogues complexes. Il est alors nécessaire de considérer des variantes.

Une première variante est la définition hiérarchique des réseaux (hierarchical STN). Un tel réseau est constitué d'un réseau principal et d'un ensemble de sous-réseaux. L'analogie est immédiate avec la notion de programme et de procédures. Les transitions se trouvent étiquetées par des sous-réseaux. On étend la définition de la validation d'une transition : on dit qu'une transition est validée si l'utilisateur donne une séquence d'actions valide pour le sous-réseau. De cette manière, le programmeur d'interfaces peut décomposer le dialogue en unités logiques (un sous-réseau par commande, des séquences communes d'actions, ...), qu'il assemble grâce aux transitions. Toutefois, il faut remarquer que la puissance de description n'augmente pas car on peut toujours remplacer l'appel du sous-réseau par le réseau correspondant et obtenir un graphe STN classique.

Par contre, les réseaux récursifs de transitions (RTN pour Recursif Transition Network) augmentent la puissance de description de ce modèle. Il s'agit de réseaux hiérarchiques dont les sous-réseaux peuvent s'appeler récursivement. L'exemple est celui de la construction d'un polygone (ligne brisée fermée) avec support de la fonction défaire (annulation du dernier point sauf le premier) :

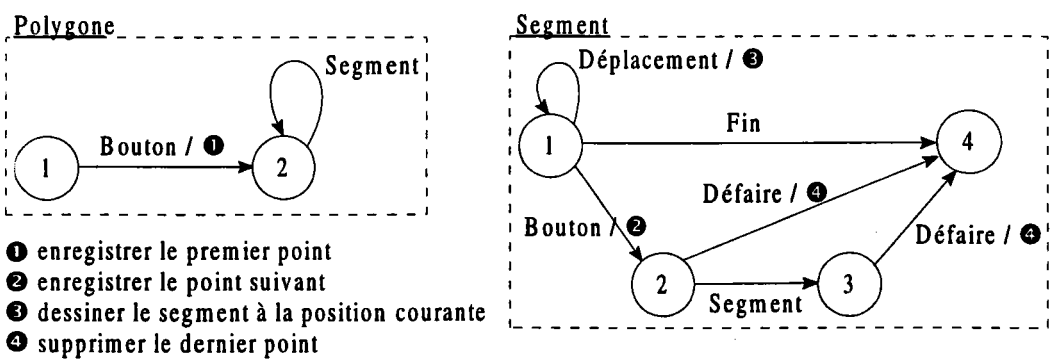
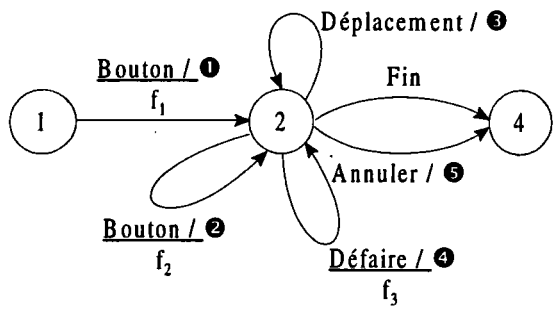


Figure I.7. *Le réseau récursif de transitions de construction d'un polygone.*

Cette action ne peut être décrite à l'aide des réseaux hiérarchiques classiques. Pourtant, bien que les réseaux récursifs autorisent la description de fonctions telles que l'annulation, on ne peut pas affirmer que ces réseaux supportent aisément la conception de dialogues contextuels. Pour cela, il faut, entre autres. qu'ils tiennent compte de l'état de l'application et des différents types d'utilisateurs. Les réseaux

augmentés de transitions (ATN pour Augmented Transition Network) sont une variante des réseaux de transitions, auxquels sont ajoutés un ensemble de registres et un ensemble de fonctions [BOR 92]. Les registres sont des variables pouvant conserver des valeurs arbitraires seulement visibles à l'intérieur du composant *Contrôle du dialogue*; l'application ne peut y accéder. Les fonctions effectuent des calculs sur les registres et changent leur valeur; elles ne peuvent accéder à l'application. Ces fonctions sont attachées aux transitions du réseau et on étend à nouveau la définition de la validation d'une transition. On dit qu'une transition peut être validée si la valeur retournée par la fonction est vraie.



❶ enregistrer le premier point
❷ enregistrer le point suivant
❸ dessiner le segment à la position courante
❹ supprimer le dernier point
❺ effacer le polygone

f_1 : nb := 1; retourner (VRAI);
f_2 : nb := nb+1; retourner (VRAI);
f_3 : si (nb = 1)
    alors retourner (FAUX);
    sinon nb := nb-1;
        retourner (VRAI);
    fsi

Figure I.8. *Le réseau augmenté de transitions de construction d'un polygone.*

Les réseaux de transitions sont faciles d'utilisation (surtout les STNs), car ils possèdent une séquence temporelle explicite contrairement aux grammaires mais ils peuvent devenir de vrais labyrinthes [MYE 89]. Le problème principal provient de l'explosion combinatoire de ces réseaux aussi bien en nombre d'états qu'en nombre de transitions. De plus, en se concentrant sur l'état du dialogue, ils insistent sur la notion de mode du système et sur la séquence de transitions d'un mode vers un autre. Ils sont donc plus adaptés pour les interfaces devant posséder plusieurs modes (un état du réseau est vraiment un mode du dialogue) et sont peu adaptés, en l'état, pour les interfaces où la sémantique joue un grand rôle pour déterminer les dialogues qui suivent.

Pour étendre l'utilisation de ces réseaux et tenter de répondre au problème de leur taille, des travaux ont été effectués par l'éclatement du dialogue en unités indépendantes. Par exemple, le dialogue d'interfaces à manipulation directe peut être décrit par un ensemble d'objets, chaque objet possédant son propre réseau de transitions et ses propres données [JAC 86].

D'autres extensions ont été apportées notamment pour la spécification des applications concurrentes. Il s'agit en particulier des réseaux de Petri [DAV 89]. Par rapport aux réseaux de transitions, ce modèle ajoute une représentation explicite des problèmes de parallélisme et de synchronisation (par exemple, la synchronisation de deux réseaux distincts exécutés en parallèle). Pour cela, un ou plusieurs jetons circulent sur le réseau pour indiquer les états actifs et le jeton avance en fonction des synchronisations. Le modèle utilisé dans le cadre des interfaces homme-machine est une version particulière des réseaux de Petri : les réseaux de Petri à structure de données [BAR 86]. Dans un tel réseau, les jetons sont remplacés par des objets. Ainsi, le marquage d'un état du réseau se fait par un objet et chaque transition

du réseau est étiquetée par un ensemble de variables. Une classe est associée à chaque état et à chaque variable; un état ou une variable ne peut recevoir que des objets de sa classe. À chaque transition, on associe une pré-condition (expression booléenne construite à partir des variables d'entrée de la transition) et une action (instruction d'affectation). Les règles d'évolution sont identiques à celles des réseaux de Petri classiques. On retrouve bien entendu les mêmes inconvénients que pour les réseaux de transitions. Ces réseaux de Petri sont utilisés pour spécifier des traitements industriels et commencent à être utilisés dans la définition des dialogues [BOR 92].

Au dela des différentes versions des réseaux de transitions, leur éclatement et l'apparition des fonctions dans les réseaux de type ATN pose tout naturellement le problème de la programmation et du langage à utiliser. Ainsi, de nombreuses études se sont directement intéressées à des langages de développement et de spécification d'interfaces tels que le modèle à événements ou les langages déclaratifs.

5.3. Le modèle à événements.

Ce modèle est à rapprocher des architectures clients/serveurs, aujourd'hui bien connues, et très utilisées dans de nombreux domaines. Il s'appuie sur le concept des événements en entrée [GRE 86]. Tous les éléments du système sont vus par le dialogue comme des générateurs d'événements. Lorsque l'utilisateur interagit avec un de ces générateurs d'événements, le générateur envoie un ou plusieurs événements.

Un événement est défini par un nom et des données associées. Par un exemple, un événement de désignation pourrait être défini par un nom 'désignation' et les coordonnées (x,y) du pointeur. Le nombre de types d'événements est arbitraire et peut être étendu pour une application donnée suivant les besoins du programmeur d'interfaces, chaque type pouvant être précisé. Ces événements, internes au composant dialogue, peuvent provenir des trois composants du système à savoir *Présentation*, *Application* et *Contrôle du dialogue* lui-même. Lorsqu'un événement est généré, il est envoyé à un ou plusieurs gestionnaires d'événements (events handlers).

Un gestionnaire d'événements est un processus capable de traiter certains types d'événements. Dans un tel système, un gestionnaire est créé par instanciation d'un patron qui en définit le comportement. Le résultat de cette création est un nom unique permettant d'y faire référence. Un patron est décomposé en plusieurs sections définissant les paramètres du gestionnaire d'événements, les variables locales, les événements traités ainsi que les réponses à apporter. Lorsqu'un gestionnaire reçoit un événement qu'il est capable de traiter, il peut avoir plusieurs réponses; il peut :

- exécuter une action. Généralement c'est la réponse à l'événement traité;
- générer un ou plusieurs événements. Cela permet la communication entre gestionnaires;
- créer de nouveaux gestionnaires. La création autorise la prise en compte de nouveaux événements en fonction de l'avancée des dialogues ou des actions de l'utilisateur (création d'un gestionnaire de désignation pour une nouvelle fenêtre, ...);

- détruire des gestionnaires (éventuellement lui-même). La destruction met fin à la prise en compte d'événements devenus inutiles au dialogue (destruction du gestionnaire de désignation d'une fenêtre allant être détruite, ...).

Cette définition de patrons est illustrée par l'exemple d'un patron pour la construction d'un rectangle avec le support de la fonction d'annulation :

```
Gestionnaire RECTANGLE
Codes Bouton,
      Déplacement,
      Défaire
Initialisation
{     état := 0;
}

Événement Déplacement
{     Si (état = 1)
      Alors
            dessiner rectangle entre
            premier et la position courante
      FinSi
}

Variables     état              (Entier)
              premier, dernier  (Point)

Événement Bouton
{     Si (état = 0)
      Alors
            premier := position courante;
            état := 1;
      Sinon
            dernier := position courante;
            créer le rectangle entre premier et dernier
            désactiver (lui-même);
      FinSi
}
Événement Défaire
{     Si (état = 0)
      Alors
            désactiver (lui-même);
      Sinon
            état := 0;
      FinSi
}
FinGestionnaire RECTANGLE;
```

Bien entendu, plusieurs gestionnaires peuvent être créés à partir du même patron et peuvent avoir des états différents. Après avoir été créé, un gestionnaire est dit actif et ce, jusqu'à sa destruction. On définit alors l'interface par l'ensemble des patrons qu'elle supporte et l'état de l'interface par l'ensemble des gestionnaires actifs. De manière conceptuelle, on considère que tous les gestionnaires d'événements s'exécutent en parallèle et traitent les événements lorsqu'ils arrivent.

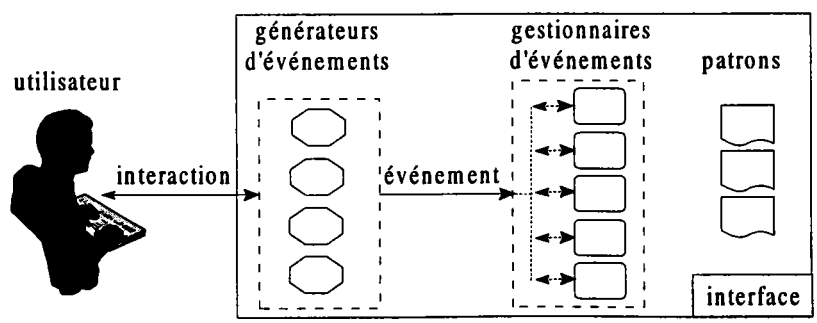


Figure I.9. *Une interface basée sur le modèle à événements.*

Au début de l'exécution du système, une instance de l'un des patrons est créée afin de servir de gestionnaire principal. Ce gestionnaire créera (directement ou indirectement) tous les autres gestionnaires de l'interface utilisateur. Comme dans le cas des grammaires, une des particularités de ce

modèle est l'abstraction que l'on impose sur le flot de données. Contrairement aux langages classiques, ce flot rendu implicite par les événements eux-mêmes est encore plus difficile à suivre et à analyser que pour les grammaires. À aucun moment, on ne peut savoir d'où viennent et où vont les données. Ce n'est pas un problème en soi (au contraire pour l'indépendance du développement c'est très important) mais cela rend difficile la compréhension d'un tel modèle pour un utilisateur non expérimenté. De plus, le nombre d'états du dialogue que l'on obtient par les créations-destructions de gestionnaires rend difficile la validation. On perd la notion d'état explicite des réseaux de transitions, ce qui ne facilite pas l'évaluation du dialogue. En fait, ce modèle peut être comparé aux règles de production, il s'appuie sur la création et la destruction d'événements. Il s'adapte donc très bien aux dialogues concurrents mais pour les dialogues séquentiels, il nécessite également une indication explicite de la séquence à l'aide de variables locales d'état ou de créations-destructions d'un nombre important de gestionnaires (dans l'exemple du rectangle, la variable état représente la séquence).

5.4. Les langages déclaratifs.

Ces langages présentent une approche totalement différente des modèles proposés jusqu'à présent. Le programmeur d'interfaces ne décrit pas le séquencement des actions à accomplir mais il définit seulement les objets et les relations qu'il veut obtenir. Par opposition aux langages impératifs traditionnels qui contiennent leur propre séquence d'actions, les langages déclaratifs décrivent ce qui va arriver plutôt que comment le réaliser. La séquence n'est pas incluse dans la description.

Les relations entre les objets peuvent être fournies de différentes façons et surtout dans n'importe quel ordre. La seule condition est que la spécification soit suffisante pour obtenir un résultat acceptable. Un système tel que COUSIN [HAY 85] fonctionne sur ce principe et fournit une interface très simple fondée sur des zones d'entrée textuelles, des menus et des zones graphiques que l'application peut utiliser comme elle le désire.

Ce modèle présente l'avantage de soulager le programmeur d'interfaces de charges inutiles. Il ne se préoccupe plus de la séquence des événements, il se focalise sur ce qu'il désire concernant les objets à fournir et à recevoir. Cependant, les outils actuels sont trop limités par les styles d'interactions fournis. Par exemple, ces systèmes ne fournissent pas encore de support pour le déplacement d'objets graphiques. Les interfaces générées sont très limitées (formulaires en règle générale) et le reste est à la charge du programmeur par un "codage à la main". Malgré ces limitations, les langages déclaratifs présentent des possibilités d'évolution impressionnantes à la condition de clairement établir le domaine auquel on s'intéresse (modélisation des objets adaptée à la génération) et le type d'interface supporté (l'interface générée répond alors aux besoins). À ce prix, nous pensons qu'il est possible de résoudre de nombreux problèmes grâce à ce modèle.

5.5. Conclusion.

Le nombre de modèles de dialogue présentés montre bien l'intérêt que leur portent les différents utilisateurs. En fait, cela illustre surtout la grande diversité des applications visées. Tous ces modèles

ont une tendance à verser dans l'universalité, ils veulent tout modéliser sans pour autant être complexes. Il faut se rendre à l'évidence, au fur et à mesure des extensions apportées, ils perdent beaucoup en facilité d'utilisation ce qu'ils gagnent en puissance d'expression.

Si l'on se place selon la puissance d'expression, le modèle à événements est certainement le modèle le plus puissant [GRE 86]. Il existe en effet des méthodes de transformation des grammaires et des réseaux de transitions vers ce modèle. A priori, tout dialogue décrit à l'aide d'une grammaire ou d'un réseau de transitions peut l'être par le modèle à événements. L'inverse n'est pas vrai. Cependant, il convient d'être prudent et de ne pas négliger la facilité d'utilisation car trop souvent l'augmentation de la puissance est synonyme d'augmentation de la complexité et d'efforts supplémentaires pour comprendre la notation. Le point essentiel que l'on doit prendre en considération est la nécessaire adéquation entre le modèle de dialogue et l'interface que l'on désire développer. Il convient de choisir le "bon" modèle pour le problème à résoudre. Le modèle à événements est peut-être le plus puissant pour la description mais le moins puissant pour l'utilisation. Par exemple, il est toujours possible de concevoir un outil supportant plusieurs modèles de dialogue tout en utilisant le modèle à événements comme modèle de base. Le programmeur d'interfaces peut ainsi choisir le modèle le plus adapté tout en profitant du modèle le plus puissant. Cette approche équilibre la balance en fonction des objectifs et des besoins, elle permet de s'adapter au programmeur et à ses désirs. De plus, il est difficile de classer les langages déclaratifs car ils peuvent être considérés comme un sur-ensemble des autres modèles.

Le choix d'un modèle est un problème en soi. Tous les modèles présentés possèdent leurs avantages qui vont de pair avec leurs défauts. Nous pensons qu'un modèle de dialogue doit être développé ou spécifié en fonction de son domaine d'application, il peut ainsi en extraire des caractéristiques que le programmeur d'interfaces n'aura plus à supporter. C'est dans ce sens que nous proposerons nos propres modèles de dialogue appliqués à la C.F.A.O. dans les chapitres 3 et 4.

6. LES OUTILS DE DÉVELOPPEMENT.

Devant l'augmentation de la demande en termes de qualités d'interface, le programmeur d'interfaces est très souvent amené à reconsidérer la dernière conception de son interface. Pourtant cette tâche est très complexe et en particulier pour les systèmes interactifs. Il s'agit de configurer un grand nombre de périphériques (clavier, souris, ...), de programmer les séquences d'actions, d'interactions (séquences de menus, ...) et de lier à l'interaction les objets calculables. Pour cela, le problème est de réduire la distance qui existe entre la pensée de conception (quoi faire) et la façon de le réaliser (comment le faire). L'objectif est de traduire la conception abstraite et les principes d'utilisation sous une forme exécutable; ce travail, réalisé par le programmeur, est souvent très fastidieux et enclin aux erreurs.

Par conséquent, il convient de fournir des outils de développement pour élever les langages de conception et d'implémentation à un niveau d'expression dans lequel le concepteur ou le programmeur peut coder l'interface directement en termes d'interactions avec les objets manipulés. Ces outils permettent de faciliter le travail par la réduction de la complexité des transformations ou par l'automatisation de certaines phases de la conception. Ils construisent des niveaux d'abstraction au

dessus des services indispensables que sont le matériel et les logiciels d'exploitation. Ils fournissent des supports pour l'implémentation d'un système interactif, à savoir comment la tâche de codage est réalisée et structurée.

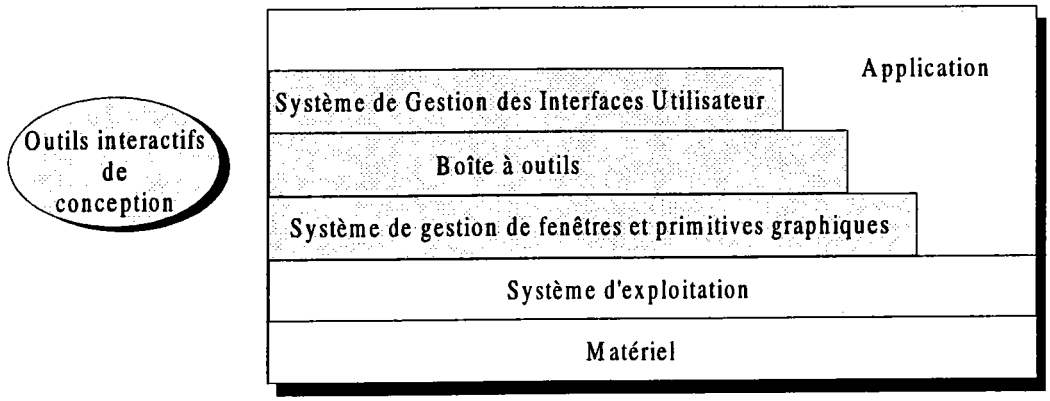


Figure I.10. *Les niveaux d'abstraction des outils de conception d'interfaces [FOL 90][MAR 91].*

Un grand nombre d'outils ont été élaborés pour rendre les interfaces utilisateur moins chers et plus faciles à développer. Ils se placent tous à des niveaux différents d'abstraction. Cela inclut les systèmes de gestion de fenêtres, les boîtes à outils et les systèmes de gestion des interfaces utilisateur. Des définitions et des analyses sur ces outils peuvent être trouvées dans [DIX 93; FOL 90; MYE 89; MYE 92c].

6.1. Les systèmes de gestion de fenêtres.

Les systèmes de gestion de fenêtres forment le niveau d'abstraction le plus bas. Ils fournissent une indépendance par rapport aux périphériques matériels tels que l'écran, le clavier, la souris, ... Ils jouent le rôle d'un gestionnaire [PAN 93]. C'est le premier niveau d'outils qui permet de simplifier et surtout de standardiser le développement des interfaces utilisateur. Le système le plus connu est certainement X développé au MIT au milieu des années 1980. Dans un tel système, le programmeur dirige ses commandes vers un terminal abstrait par l'intermédiaire d'un langage générique qui est traduit dans un langage compréhensible par les périphériques de la machine hôte.

Pour un terminal abstrait, le langage générique est appelé modèle image. Il existe de multiples modèles images, on peut citer notamment le modèles pixels, GKS, PHIGS et PostScript. À l'origine, ces modèles se destinaient essentiellement aux sorties mais depuis ils supportent également les entrées de l'utilisateur. Le modèle pixels sert pour l'entrée des coordonnées (à charge de l'application de gérer les éventuels traitements sémantiques associés) alors que les modèles PHIGS et GKS ont été enrichis pour supporter un modèle d'entrée plus explicite [BET 88; HAR 91]. Comme on le voit, les modèles images prennent en charge de plus en plus de tâches dévolues au composant *Présentation* et facilitent ainsi les interactions complexes. On trouve notamment la projection et le partage des données avec l'application. Par la projection, la partie applicative communique une vue partielle des données (par exemple GKS manipule un ensemble de segments pour les désignations). Concernant le partage des données, le système de gestion de fenêtres manipule une structure de données qui est éditable par

l'application (par exemple PHIGS fournit un protocole pour manipuler les données).

Ces systèmes forment un environnement central à la fois pour le programmeur et pour l'utilisateur en autorisant une simple station de travail à supporter simultanément de multiples dialogues. En effet, les ressources matérielles peuvent être partagées par plusieurs instances d'un terminal abstrait. Chacun de ces terminaux abstraits réagit comme un processus indépendant et est programmé comme tel. Des protocoles internes d'échange d'informations sont intégrés (ICCCM pour le système X) sous la responsabilité du système.

La prise en compte des entrées de l'utilisateur se fait par l'intermédiaire d'événements générés par le système. Le contrôle de ces événements peut être interne à l'application ou interne au système. Dans le premier cas, l'application lit les événements et détermine sa réponse en fonction de l'événement. Il est nécessaire de contrôler tous les événements même ceux sans intérêt (on parle de boucle lecture-évaluation). Dans le second cas, un notifieur interne au système soulage l'application de la gestion de tous les événements et prévient l'application lorsqu'un événement qui l'intéresse survient. Cette approche est plus souple mais pose certains problèmes pour les dialogues contextuels où il faut reconfigurer entièrement le notifieur (modification des événements intéressants). Très souvent les applications utilisent les deux types de contrôle selon les besoins.

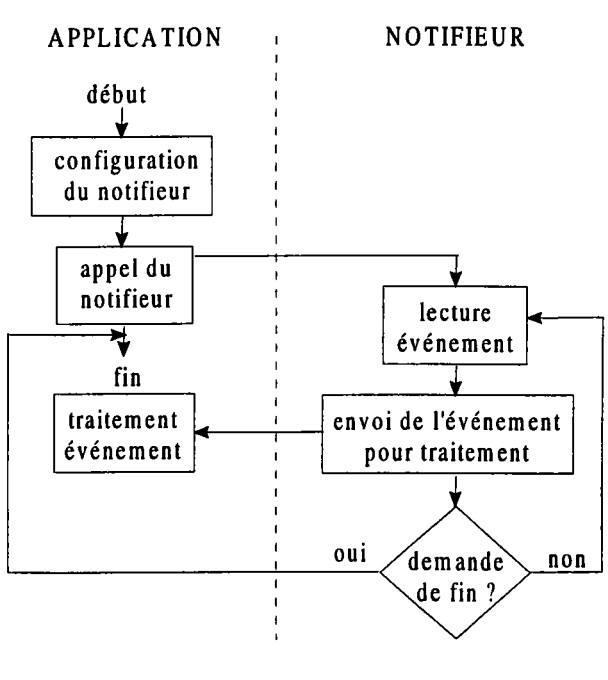


Figure I.11. *Le notifieur d'événements.*

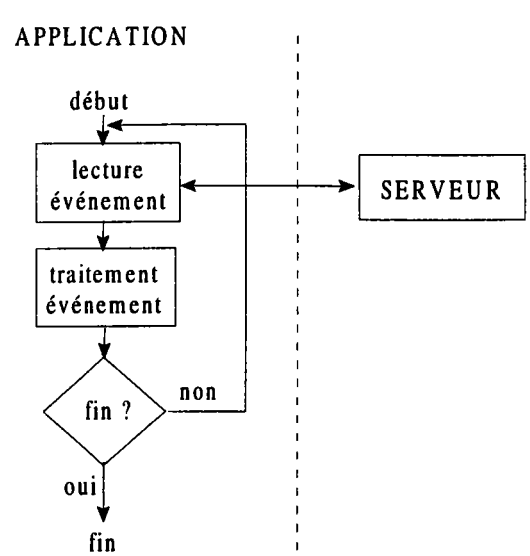


Figure I.12. *La boucle lecture-évaluation.*

Les systèmes de gestion de fenêtres fournissent les caractéristiques les plus importantes des interfaces modernes. En particulier, les applications peuvent afficher leurs résultats dans des zones différentes de l'écran, agrandir des zones pour lever des ambiguïtés, ... Il faut remarquer que ces services sont réalisés à un niveau très bas qui impose l'écriture d'une grande quantité de code et surtout avec une grande complexité d'utilisation. Ils sont accessibles à travers des centaines de primitives que le programmeur doit connaître pour pouvoir les utiliser. De nombreux travaux sont en cours sur la normalisation de ces

systèmes de gestion de fenêtres (plateformes communes de développement) mais aussi sur leur ergonomie [RUS 93]. Leur difficulté d'apprentissage a conduit tout naturellement à l'introduction des boîtes à outils qui sont destinées à en faciliter l'accès.

6.2. Les boîtes à outils.

L'objectif des boîtes à outils est de fournir une abstraction de programmation plus importante pour construire des interfaces utilisateur en minimisant la programmation. Elles sont constituées d'une bibliothèque d'éléments logiques prêts à utiliser. Chaque élément peut être considéré comme une fenêtre (window) et un outil (gadget) d'où le nom le plus fréquent : un widget (window-gadget). Chaque widget représente une façon d'utiliser un périphérique physique (souris, clavier, ...) pour donner une information (commande, nombre, ...) alors qu'un écho se fait à l'écran. Ces widgets ont un comportement spécifique pré-défini que l'on peut modifier pour une situation particulière (changer le nom d'un bouton, ...). La plupart des systèmes de gestion de fenêtres fournissent une boîte à outils constituée de widgets tels que des menus, des ascenseurs, ...

Le programmeur utilise une boîte à outils en écrivant du code dans un langage de programmation classique ou par l'intermédiaire d'un outil graphique (appelé constructeur d'interfaces). La description des comportements des objets se fait au même niveau que l'utilisateur, le programmeur perçoit le widget de la même façon que l'utilisateur (entrées/sorties). Il existe deux familles de boîtes à outils, les boîtes à outils conventionnelles et les boîtes à outils orientées objets. La première famille se résume à une collection de procédures qui peuvent être appelées par les programmes d'application (par exemple XVIEW sur station SUN). La seconde famille intègre la notion d'héritage et est constituée d'une hiérarchie de widgets, chaque widget pouvant être aisément adaptée ou étendue (par exemple UIT sous X). Il est à noter que les boîtes à outils orientées objets ne sont pas nécessairement accessibles à travers un langage objets.

```
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>

Frame frame; /* variable de fenêtre */

void quitter ()
{ xv_destroy_safe (frame); }

main (argc, argv)
int argc;
char *argv[];
{
Panel panel;

xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
frame = (Frame) xv_create (NULL,
                FRAME, FRAME_LABEL, "Quitter",XV_WIDTH, 200, XV_HEIGHT, 100, NULL);
panel = (Panel) xv_create (frame, PANEL, NULL);
(void) xv_create(panel, PANEL_BUTTON,
                PANEL_LABEL_STRING, "Quitter", PANEL_NOTIFY_PROC, quitter, NULL);

xv_main_loop (frame);
exit (0);
}
```

L'utilisation d'une boîte à outils offre non seulement une bonne portabilité, grâce aux concepts introduits, mais aussi une grande consistance des interfaces produites. Toutes les entrées et sorties

fournissent un comportement similaire à un ensemble de widgets (tous les boutons ont le même comportement pour l'utilisateur). Le programmeur dispose d'une grande souplesse d'utilisation, il a le choix entre le plein contrôle ou l'utilisation de services plus évolués. Cependant, cette liberté entraîne un risque important de mauvaise décomposition; sans structuration, on aboutit très souvent à de mauvaises architectures logicielles. Leur apprentissage est très difficile car il n'est pas toujours clair de déterminer quels éléments utiliser pour réaliser ce que l'on désire. Le nombre de primitives est très important et les mécanismes qui assurent la gestion de l'ensemble des widgets sont difficiles à réaliser (même des programmeurs expérimentés ont du mal à les utiliser). De plus, les styles de widgets sont très souvent limités.

Pour répondre à la difficulté d'utilisation et aux styles limités, des outils particuliers sont apparus. On peut trouver notamment des ateliers de construction de widgets [ZEL 93] et surtout le système Peridot (Programming by Example for Real-time Interface Design Obviating Typing) [MYE 90a]. Peridot utilise une approche différente des boîtes à outils traditionnelles. Au lieu de fournir une bibliothèque de base, le concepteur crée interactivement ses widgets aussi bien du point de vue de l'apparence que du fonctionnement ("look and feel"). Pour cela, le système utilise la programmation par l'exemple. À partir de l'exemple, le système déduit des relations (contraintes) qui permettront aux instances du widget d'avoir le comportement et l'apparence spécifiés par le concepteur.

En fait, par rapport aux systèmes de gestion de fenêtres, les boîtes à outils ne fournissent pas plus de support dans la conception des interfaces et notamment, la spécification du séquencement et du contrôle du dialogue. Le contrôle est le domaine privilégié des systèmes de gestion des interfaces utilisateur.

6.3. Les systèmes d'aide au développement.

Un système de développement d'interfaces utilisateur est un environnement complet et intégré d'outils de développement et de prototypage des interfaces utilisateur. Cet environnement doit assister le programmeur d'interfaces dans sa tâche de création, de maintenance et de modification des interfaces en accord avec les besoins de l'utilisateur. Il aide donc aussi bien dans la phase de conception que dans la phase d'implémentation. Pour cela, il gère tous les aspects de l'interface, les parties visibles de l'affichage et tous les aspects du dialogue entre l'utilisateur et l'application en s'appuyant sur un certain nombre de techniques pour gérer, implémenter et évaluer l'exécution d'un environnement interactif [MYE 89]. En particulier, il est destiné à la création et à la gestion des éléments de bas niveau [MYE 90a] et notamment la définition et le contrôle des relations existantes entre les objets de présentation d'une boîte à outils et leur sémantique dans l'application. En plus des possibilités offertes par une boîte à outils, un tel système se doit de soulager au mieux le programmeur d'interfaces dans la définition de certains comportements tels que la gestion des erreurs, les abandons, les affichages, les aides, ...

Dans cette catégorie, il convient de distinguer deux sortes de systèmes :

- les Systèmes de Développement des Interfaces Utilisateur (SDIU ou UIDS pour User Interface Development System).
- les Systèmes de Gestion des Interfaces Utilisateur (SGIU ou UIMS pour User Interface Management System);

Les SDIUs sont des environnements plus particulièrement intéressés par la phase qui précède la gestion de l'exécution. Ils facilitent la conception de l'interface mais pas nécessairement pour les interactions à l'exécution. En général, ils demandent au programmeur de décrire le niveau sémantique en termes de fonctionnalités à travers des objets et des opérations [PRE 94].

Par contre, les SGIUs sont des systèmes de support à l'exécution, à la conception, à la spécification et à l'évaluation [FAC 92]. En plus de la définition de l'interface, ils prennent en charge son fonctionnement. Ils se focalisent généralement sur les problèmes rencontrés lors de l'exécution de l'interface car ils sont responsables des interactions [PRE 94]. Ils demandent au programmeur de définir principalement les menus, les commandes et les règles de séquencement des actions et des interactions.

Il est possible de classer les SGIUs selon les modèles qu'ils utilisent (modèles d'architecture et de dialogue associés). En effet, l'implémentation est fortement influencée par le modèle d'architecture utilisé car le système fournit le plus souvent des outils de conception et d'implémentation pour chaque composant du système, sa structure suit le modèle choisi [GRE 86]. Par exemple GARNET [MYE 92d], articulé autour d'objets régis par des contraintes, fournit un outil de construction de widgets (Lapidary), un gestionnaire de contraintes (C32) et un outil de génération de boîtes de dialogue (Jade). Par conséquent, si les modèles de base sont bien définis, les outils correspondants sont aisés à concevoir. Un programmeur d'interfaces a toute facilité à utiliser des outils parfaitement intégrés entre eux. Il est donc essentiel avant la conception d'un SGIU, de bien définir son modèle qui est le cœur de son fonctionnement [JAC 86].

Jusqu'à présent, les SGIUs se sont concentrés sur la gestion du séquencement de l'interface utilisateur (syntaxe du dialogue). Les modèles de dialogue servent de base à cette définition pour indiquer quelles opérations sont autorisées à un instant donné et les traitements à effectuer après chaque opération. Les éditeurs interactifs ainsi développés aident à combiner et séquencer les techniques d'interaction, mais, de plus en plus souvent, le programmeur d'interfaces demande des outils de très haut niveau aussi bien du point de vue fonctionnel que conceptuel. Actuellement, les travaux s'orientent donc vers le composant *Interface avec l'application* du modèle Seeheim [JOH 93]. Le fait de mieux connaître les relations entre l'interface et l'application poussent les recherches vers la décomposition et le paramétrage de ce composant afin de disposer de meilleures connaissances sur les objets ce qui est primordial pour la manipulation directe. On parle parfois de systèmes post-SGIU, on peut citer Nephew [SZE 89] et UIDE [FOL 93; GIE 92]. L'évolution se poursuit tout naturellement vers la génération automatique de l'interface utilisateur à partir du modèle de données fournit par le programmeur d'applications [DEN 92; JAN 93; KIM 93; MOR 94]. Les systèmes Unidraw [VLI 90] et

Workspaces [OLS 92] sont des exemples significatifs de cette approche.

Par l'utilisation de tels sytèmes, le programmeur crée des interfaces le plus souvent sans programmation et sans avoir à apprendre des détails sur la boîte à outils sous-jacente. Ils représentent a priori le niveau final d'outils d'aide au développement. Ils fournissent un support beaucoup plus large qu'une boîte à outils, une infrastructure flexible, des composants de haut niveau et très souvent des langages de définition bien intégrés dans l'environnement [JOH 93]. Cependant, il ne faut pas oublier que ces outils sont difficiles à utiliser car ils nécessitent souvent d'apprendre un nouveau langage dédié (graphique ou non). De plus, la structure pauvre de ces langages qui utilisent de nombreuses variables globales et le flot de contrôle global rendent difficile la compréhension et l'édition des spécifications.

6.4. Conclusion.

Comme nous avons pu le constater, l'apparition des différents outils de développement a suivi une démarche ascendante. Du niveau le plus bas, que sont les serveurs graphiques, au niveau le plus haut avec les SGIUs, l'objectif a toujours été la baisse des coûts de développement, l'augmentation de la rapidité d'utilisation et de la fiabilité des interfaces réalisées. Progressivement, ces outils prennent en charge de plus en plus de tâches, que ce soit la présentation, les interactions, ... Des études montrent qu'ils sont très utilisés à des degrés différents : 74% des personnes ayant à développer une interface utilisent un outil de développement avec des gains considérables en productivité. Même si 34% se contentent d'une boîte à outils, il semble bien que la demande pour des outils de haut niveau soit très importante.

Il me semble que l'utilisation d'un SGIU est primordiale, essentiellement grâce à l'intégration des outils qu'il autorise. Un tel système fédère tous les intervenants et notamment les programmeurs, les graphistes, les ergonomes, les cogniticiens, ... Cependant l'éternel dilemme se pose entre la facilité d'utilisation et la puissance des techniques implantées. Trop souvent les techniques d'interaction ne sont pas appropriées pour l'intervenant le plus intéressé : l'utilisateur. Il est très difficile de trouver un équilibre entre les besoins d'un programmeur d'interfaces et ceux de l'utilisateur. Pourtant, un bon équilibre doit venir de l'intégration de l'utilisateur dans la phase de conception de toutes les interfaces et ce, dans un sens plus large : les tâches de conception doivent être bien guidées et satisfaire les besoins de l'utilisateur.

De plus, lors des phases de développement, l'utilisateur est encore mal intégré voire pas du tout. Pourtant le placer au centre du développement (user-centered) doit prévenir des mauvaises interfaces et encourager une "bonne" conception [JOH 93]. Il est très important de fournir en premier lieu des règles adaptées (guidelines) pour garantir la consistance des interfaces, règles édictées par des utilisateurs pour des utilisateurs. De cette façon, l'utilisateur pourra s'appuyer sur des stratégies similaires entre les différentes applications grâce à des concepts communs.

7. CONCLUSION.

Devant l'accroissement de la part dévolue au dialogue dans le développement d'une application, il faut considérer la conception des interfaces utilisateur comme une part intégrante du processus de conception d'un système. Nous nous sommes attachés à clairement distinguer les modèles qui interviennent dans cette conception. En particulier, nous avons différencié le modèle d'interaction du modèle de dialogue. Alors que le modèle d'interaction définit la communication entre l'utilisateur et la machine (l'interaction homme-machine), le modèle de dialogue décrit la suite d'interactions à mettre en œuvre pour réaliser une action (le dialogue homme-machine). La séparation de l'interface de la partie application proposée par les modèles d'architecture est en fait une extension normale du génie logiciel. Cette spécification de l'interface doit se faire simplement et sans programmation; un utilisateur non informaticien utilisant un langage non conventionnel et des outils devrait pouvoir créer le dialogue de son système.

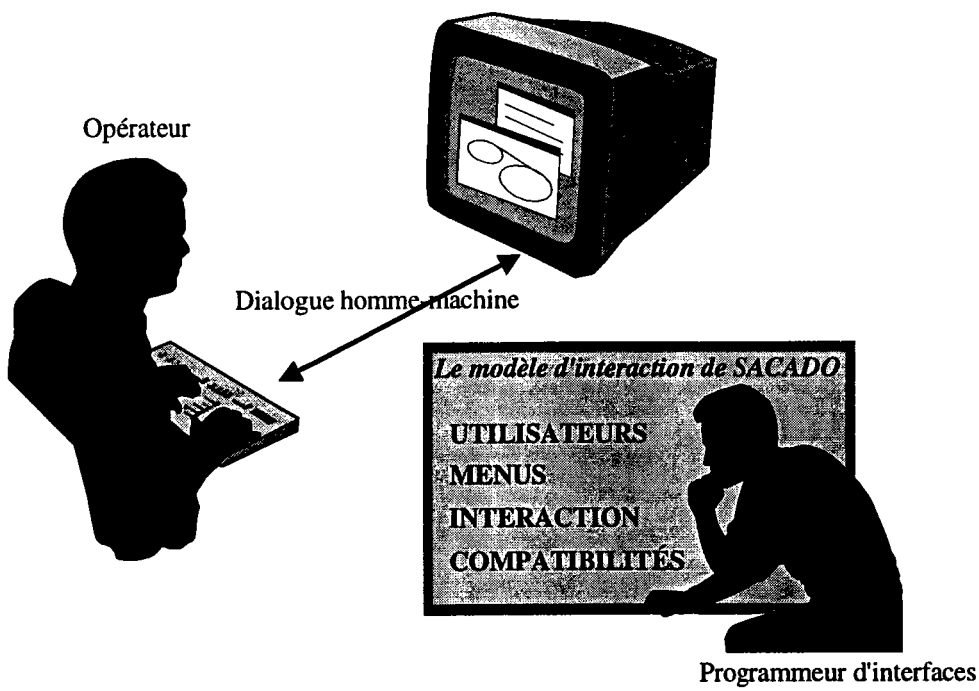
Cependant, ce chapitre montre qu'un grand nombre de paramètres sont à prendre en compte, ne serait-ce que la complexité de l'interlocuteur lui-même. La recherche de la généricité est l'une des difficultés majeures des outils de conception actuels. Tous les modèles développés se veulent les plus généraux possibles et englobent le plus large champ d'applications possible. Par ce fait, ils désirent avoir la plus grande puissance de description, à défaut de la plus grande facilité d'utilisation. L'apparition de systèmes à manipulation directe a montré que l'on pouvait développer des systèmes utilisables sur des domaines relativement restreints et bien connus.

Malgré tous les travaux réalisés sur les interfaces utilisateur, aucun des modèles présentés ne fait état du type d'applications que l'on peut développer. Le cas des modèles de dialogue est édifiant. Tous les modèles sont étendus au fur et à mesure des besoins pour aboutir finalement à des modèles extrêmement complexes et utilisables seulement par des utilisateurs expérimentés.

Dans les chapitres suivants, nous nous proposons d'apporter une réponse au développement des systèmes de C.F.A.O. Après une étude des difficultés inhérentes à ce domaine, nous détaillons une approche originale basée sur une primitive unique d'interaction. En s'appuyant sur un style d'interactions utilisant des menus et des compatibilités (liens logiques entre les interactions et les menus), nous permettons au programmeur d'interfaces de décrire de façon structurée et contextuelle la hiérarchie de tâches qu'il désire autoriser. À partir de ce modèle d'interaction, nous introduisons deux modèles de dialogue (l'un textuel, l'autre graphique) adaptés à ce domaine et qui répondent aux problèmes posés par certains modèles existants. Nous complétons nos travaux par une étude d'un langage de type déclaratif qui permet la génération du dialogue à partir du modèle des objets manipulés.

CHAPITRE 2.

UN MODÈLE D'INTERACTION POUR LA C.F.A.O.



1. INTRODUCTION.

La C.F.A.O. (Conception et Fabrication assistée par Ordinateur) est un domaine des applications graphiques interactives. La plupart des systèmes de C.F.A.O. manipulent une grande quantité d'objets graphiques en constante évolution et régie, le plus souvent, par de nombreuses relations.

Dans ce chapitre, nous étudions un modèle d'interaction qui offre une grande facilité de description tout en fournissant un contrôle sémantique des fils d'activités multiples du dialogue. Il s'agit de faciliter la très forte interactivité des outils de C.F.A.O. tout en vérifiant que cette interactivité se fait en accord avec la sémantique de l'application.

2. LA C.F.A.O.

Les outils de C.F.A.O. sont le résultat de l'intégration d'outils de C.A.O. (Conception Assistée par Ordinateur) et d'outils de F.A.O. (Fabrication Assistée par Ordinateur) afin d'augmenter la productivité se situant entre la conception et la chaîne de fabrication :

- la C.A.O. représente l'ensemble des aides informatiques que l'on fournit aux bureaux d'études et de méthodes depuis l'élaboration du cahier des charges jusqu'à l'établissement des documents nécessaires;
- la F.A.O. définit l'utilisation de l'informatique pour planifier, gérer et contrôler les opérations de fabrication.

Notre propos n'est pas de présenter la C.F.A.O. de manière exhaustive mais d'en étudier quelques aspects essentiels qui permettront d'une part d'avoir une vue d'ensemble des difficultés et d'autre part de mieux situer notre travail.

Aujourd'hui, la C.F.A.O. est l'instrument de base de la production industrielle; son but est de définir un modèle de l'objet conçu assez complet et cohérent pour pouvoir être utilisé comme un prototype virtuel dans des essais de différentes sortes : visualisation, simulation, fabrication, ... Deux notions très importantes apparaissent rapidement : le modèle (de l'objet) et le dialogue homme-machine (pour accéder au modèle).

Avant de détailler ce que représentent le modèle (de l'objet) et le dialogue homme-machine pour un système de C.F.A.O., il nous semble important de définir clairement les différents utilisateurs qui interviennent sur un tel système. Nous distinguons essentiellement trois familles d'utilisateurs, à savoir :

- l'opérateur (ou utilisateur final);
- le programmeur d'interfaces;
- le programmeur d'applications et de modèles.

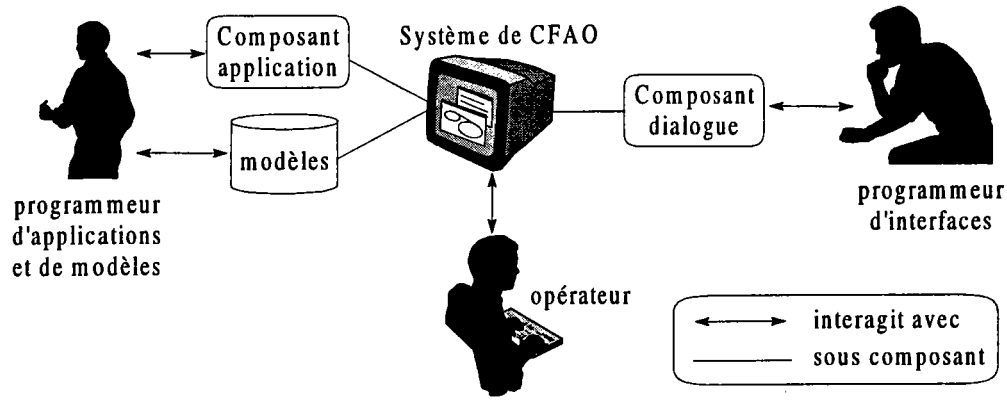


Figure II.1. *Les trois utilisateurs d'un système de C.F.A.O.*

L'opérateur utilise l'application dont les fonctionnalités ont été au préalable mises en œuvre par le programmeur d'applications. L'enchaînement des différentes fonctionnalités et le dialogue associé ont été développés par le programmeur d'interfaces. Dans de nombreux cas, le programmeur d'interfaces peut être l'opérateur lui-même.

Le modèle est une représentation informatique de l'objet conçu ou en cours de conception. Il constitue le cœur du système de C.F.A.O. En fait, un tel système est défini à partir d'une multitude de modèles dits applicatifs ayant chacun un rôle particulier :

- définition des propriétés géométriques des objets (B-REP, CSG, ...);
- définition des données propres à la représentation visuelle des objets (modèles de visualisation associés aux techniques de visualisation, ...);
- définition des données nécessaires à la fabrication (modèle de commande numérique, ...)

Le rôle de ces modèles s'apparente à celui d'une authentique maquette virtuelle où il sera possible d'effectuer des opérations et des essais qui relèvent du prototypage. Ils sont tous définis à partir d'informations extraites d'un modèle central que nous appelons modèle générique. Nous définissons alors la localisation comme le passage du modèle générique vers un modèle applicatif. L'opération inverse est appelée globalisation.

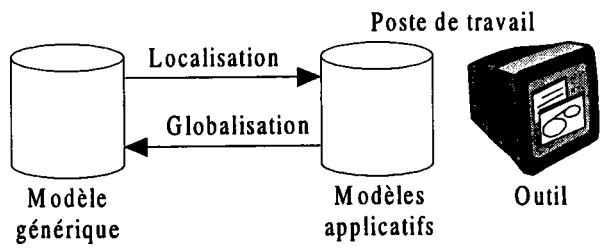


Figure II.2. *La localisation et la globalisation.*

Nous verrons dans le chapitre 4 toute l'importance du modèle générique dans la définition du dialogue-machine. Nous proposons d'ajouter des informations au modèle générique pour déduire en tout ou partie, après une localisation, le modèle de dialogue de l'application que nous considérons comme un modèle applicatif à part entière (visualisation de l'objet à travers le dialogue).

Le dialogue homme-machine d'un système de C.F.A.O. doit permettre à l'opérateur de construire l'objet désiré. Malheureusement (mais heureusement pour la créativité), un processus de conception de la plupart des objets ne peut être entièrement formalisé et la mise en place d'un dialogue est nécessaire. Cependant, la réalisation d'un tel système se heurte à de nombreuses difficultés inhérentes à ce domaine et aux modèles manipulés. De ce fait, les dialogues deviennent complexes pour plusieurs raisons :

- les modèles construits sont généralement très fortement structurés avec de fortes contraintes entre les objets (contraintes de tangence, de dimensionnement, d'ingénierie, ...). Même lors de modifications mineures du modèle, le dialogue doit prendre en compte un grand nombre de dépendances;
- les dialogues sont nombreux car les traitements font intervenir un grand nombre d'objets pour mettre en place ces contraintes. On parle d'interactions multi-objets (construction d'un segment tangent à deux cercles, ...);
- les entrées de l'opérateur ne sont plus seulement numériques ou alphanumériques mais elles deviennent grapho-numériques [GAR 91]. Cela permet la définition d'un opérande à l'aide d'une expression qui fait intervenir à la fois des données numériques, alphanumériques et graphiques (par exemple l'origine d'un segment est égale au milieu d'un autre segment, ...);
- les réponses de l'opérateur sont soumises à de nombreuses ambiguïtés qu'il faut sans cesse lever. La structure même des données ne permet pas de déterminer quel objet est désigné par l'opérateur. Dans la Figure II.3, le système peut interpréter la désignation de trois façons : le contour, le segment ou un couple de coordonnées. Seul le contexte de l'interaction peut aider le dialogue à déterminer ce que l'opérateur a voulu désigner.

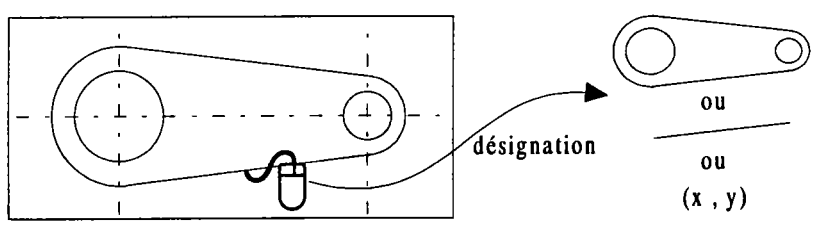


Figure II.3. *Les ambiguïtés d'une désignation.*

- des informations implicites sont à considérer. La désignation elle-même est d'une importance capitale, sa localisation sera utilisée par l'action à laquelle l'objet est destiné. La figure II.4 montre comment le système détermine quelle tangence calculer en fonction de la position de désignation.

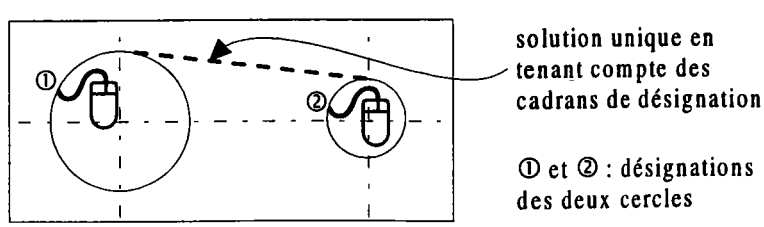


Figure II.4. *Les informations implicites d'une désignation.*

Toutes ces difficultés montrent la nécessité de définir un modèle d'interaction pour la C.F.A.O. dont l'objectif est de préciser les modèles existants pour prendre en compte les caractéristiques de ce domaine. Nous décrivons donc dans ce chapitre, le système SACADO (Système Adaptatif de Conception et d'Aide au Développement par Ordinateur) développé au Laboratoire de Recherche en Informatique de Metz [GAR 88] [GAR 93]. C'est une plate-forme de développement pour systèmes de C.F.A.O. qui sert de système fédérateur aux équipes de recherche du laboratoire pour le test et la validation de leurs travaux. Notre objectif est de réunir les trois utilisateurs, que l'opérateur puisse utiliser le système à l'aide d'un dialogue proche de son langage métier (en particulier le dessin), que le programmeur d'interfaces puisse décrire l'architecture du système (assemblage de modules et règles de déroulement lors d'une interaction) sans utiliser un langage informatique et que le programmeur d'applications (que l'on peut considérer comme un sous-traitant du programmeur d'interfaces) puisse développer les modules informatiques, selon les spécifications du programmeur d'interfaces.

À ce titre, SACADO entre dans la catégorie des systèmes de gestion d'interfaces utilisateur (SGIUs) car il fournit les outils de développement et supporte l'exécution de l'application ainsi décrite. C'est à la fois un noyau de développement et une application à part entière (l'ajout de nouvelles fonctionnalités ou d'un nouveau modèle permet de définir une nouvelle application).

Comme noyau de développement, SACADO s'appuie sur deux générateurs (outils) pour créer et compléter une application :

- le générateur de dialogue et d'architecture qui a pour but la description du dialogue et de l'architecture logicielle de l'application. C'est ce générateur que nous étudions dans les chapitres 2 et 3;
- le générateur de modèles qui permet la mise en œuvre des modèles pour une application donnée. C'est ce générateur que nous aborderons plus précisément dans le chapitre 4 en mettant en avant son importance dans la spécification du dialogue.

Les travaux qui ont conduit à la définition du générateur de dialogue et d'architecture avaient pour contraintes de :

- fournir un modèle d'interaction autorisant une grande liberté à l'opérateur tout en donnant au programmeur d'interfaces les moyens de contrôler la validité des interactions;
- fournir un modèle de dialogue permettant au programmeur d'interfaces de décrire l'architecture de l'application à travers le contexte de l'interaction. Nous proposons deux modèles de dialogue destinés à notre modèle d'interaction dans le chapitre 3.

Le modèle d'interaction que nous décrivons dans ce chapitre intègre un certain nombre de concepts liés à l'ergonomie autorisant la description de l'ensemble des fonctionnalités dont l'opérateur a besoin, ainsi qu'une organisation de celles-ci la plus proche possible du monde de l'opérateur et de son mode de pensée. Son rôle est de n'imposer qu'un minimum de contraintes à l'opérateur et d'interpréter au mieux ses intentions. Il s'agit de rester en accord avec sa logique d'utilisation. Pour cela, il est

nécessaire de fournir des principes de dialogue dont la généralité permettra de rester indépendant des applications et garantira la consistance des applications développées [PAN 93] :

- consistance sémantique. Un opérateur doit savoir utiliser une application sans connaissance préalable, les stratégies à mettre en œuvre pour satisfaire un objectif doivent être similaires;
- consistance syntaxique. Les mêmes termes doivent être utilisés pour réaliser deux actions identiques, les concepts doivent être communs;
- consistance lexicale. La même séquence d'interactions doit être utilisée pour réaliser la même fonction.

Par des concepts simples mais puissants comme la gestion contextuelle des fils d'activités multiples, nous offrons la possibilité de paramétrer les interactions de façon très précise selon les besoins. La finalité est qu'un opérateur (expert dans son domaine mais pas en C.F.A.O.) puisse participer au développement de l'architecture du système de C.F.A.O. et même décrire entièrement l'interface utilisateur. Il est possible pour un opérateur de définir le système en sous-traitant à un programmeur expert (programmeur d'applications) quelques fonctions complexes qui ne sont pas liées au dialogue.

3. LA DÉFINITION STATIQUE.

Dans ce paragraphe, nous présentons les concepts de base pour la construction d'une application SACADO. Nous décrivons en particulier le style d'interaction que nous avons choisi en fonction des interactions requises en C.F.A.O. Nous introduisons les actions globales liées aux menus qui constituent une action de l'application développée, action que nous décomposerons en un graphe d'actions interactives (échanges avec l'opérateur) et non interactives (modules applicatifs). Pour les actions interactives, nous avons développé une primitive de dialogue unique, appelée INTERACTION qui favorise l'opérateur dans la mesure où, quel que soit le dialogue engagé, sa réponse se fait toujours selon le même schéma. Ainsi, l'ensemble des menus associé aux actions globales représentées par un graphe d'actions représente ce que l'on appelle la définition statique d'une application SACADO.

3.1. Les menus.

Un système de C.F.A.O. doit offrir à l'opérateur la possibilité de réaliser une maquette virtuelle de l'objet qu'il désire. Mais la nature complexe des relations à mettre en place impose l'utilisation d'un dialogue qui repose sur de nombreuses manipulations d'objets. De nombreux travaux et réalisations ont porté sur la prise en compte de ces relations par manipulation directe : par exemple la détection de la tangence par proximité, ... On peut citer notamment le logiciel ASHLAR VELLUM®. Toutefois dans certaines situations, un dialogue de type verbe-objets est bien mieux adapté, l'opérateur indique alors ce qu'il veut obtenir et chaque entrée (objets) correspond aux opérandes de l'action (verbe) [VITUID]. Par ailleurs, cette approche n'est pas contradictoire avec la manipulation directe, dans la mesure où il est toujours possible d'intégrer des actions de détection dans une telle architecture. Une maquette basée sur la détection a été développée et a montré la complémentarité des deux types d'interaction [GAR 95a].

Comme nous l'avons vu dans le chapitre précédent, les styles d'interaction les mieux adaptés pour les interactions de type verbe-objets sont les menus ou les langages de commandes. Dans SACADO, nous définissons donc l'ensemble des fonctionnalités offertes par un ensemble de domaines, chaque domaine étant constitué d'un ensemble de menus.

Un domaine particulier et unique que nous appelons domaine propre définit les fonctionnalités de base de l'application. Il est accessible à l'opérateur dès le début de la session de travail. Les autres domaines que nous appelons des domaines annexes regroupent des fonctionnalités particulières auxquelles l'opérateur a accès dans un contexte particulier pour des besoins ponctuels à définir. Ces domaines ne sont accessibles qu'à travers des menus du domaine propre.

On a vu que l'opérateur décompose de manière naturelle le but à atteindre en sous-buts jusqu'à obtenir des opérations élémentaires [CAI 92]. De ce fait, les menus d'un domaine sont organisés selon une structure hiérarchique. Il s'agit d'un arbre n-aire de menus dans lequel on trouve deux sortes de menus :

- des menus non terminaux. Ils traduisent un concept général (même famille) dont les actions plus précises sont définies dans les menus fils;
- des menus terminaux. Ils correspondent à une fonctionnalité élémentaire. Ces menus ne sont pas directement liés à un appel de primitives mais à une action globale décrite par la suite.

Dans un contexte donné, un menu qu'il soit terminal ou non est :

- actif. L'opérateur peut le sélectionner. Si le menu est terminal, son action globale associée est exécutée. Dans le cas contraire ce sont ses fils qui deviennent actifs;
- inactif. L'opérateur ne peut pas le sélectionner. Il ne sera donc pas affiché ou il le sera sous une forme mettant en évidence qu'il n'est pas actif. Le fait de l'afficher permet à l'opérateur de se faire une idée des fonctionnalités d'une application sans pour autant pouvoir les utiliser.

On appelle MENUS l'ensemble des menus définis pour une application donnée. Le contexte des menus qui définit le contexte du système est défini par un ensemble de couple (menu, état) tel que menu $\in$ MENUS et état $\in$ (actif, inactif). Dans cet ensemble, tous les menus doivent être représentés :

$$\text{ContexteMenus} = \{ (\text{menu}, \text{état}) \mid \text{menu} \in \text{MENUS et état} \in (\text{actif}, \text{inactif}) \}$$
$$\text{et } \forall \text{ menu} \in \text{MENUS}, \exists (\text{menu}, \text{état}) \in \text{ContexteMenus}$$

3.2. Les actions globales.

À chaque menu terminal des domaines de l'application, on associe une action globale. C'est un concept englobant l'ensemble des traitements à effectuer pour réaliser une fonctionnalité de l'application. Une action globale comporte généralement un résultat : l'objet créé, l'objet modifié, ... L'ensemble des actions globales définies pour une application donnée est noté ACTIONS GLOBALES. On peut considérer une action globale comme un graphe d'actions que l'opérateur parcourt pour indiquer les données nécessaires et visualiser le résultat de ses actions.

La stratégie est de séparer les dialogues en deux parties :

- une partie indépendante de l'application considérée (fonctionnalités générales);
- une partie dépendante (règles d'utilisation des fonctionnalités).

Les modules de la partie indépendante sont développés au préalable comme base de SACADO. Le programmeur d'interfaces doit alors décrire la partie dépendante pour concevoir l'interface de l'application considérée. Ainsi, dans le graphe d'une action globale, on distingue deux sortes d'actions :

- les actions non interactives;
- les actions interactives.

Les actions non interactives représentent des traitements élémentaires pour le programmeur d'interfaces et sont décrits plus particulièrement par le programmeur d'applications (partie indépendante). Une telle action ne doit pas comporter de dialogue avec l'opérateur.

Les actions interactives sont chargées des échanges de données entre le système et l'opérateur. Il s'agit d'informer l'opérateur des données nécessaires et de lui fournir les moyens d'y répondre (désignation d'un objet, construction de l'objet, ...).

Par exemple, la construction d'un segment de droite fait appel à deux actions interactives, l'une pour le point origine et l'autre pour le point extrémité, et à une action non interactive pour la construction du segment passant par ces deux points. Le programmeur d'interfaces crée de cette manière le graphe du dialogue que l'opérateur doit parcourir pour réaliser son but. L'opérateur construit alors une suite d'actions $(a_1, \ldots, a_n)$ en fournissant des données, ce qui correspond à un chemin à travers le graphe du dialogue.

3.3. Les actions interactives.

Dans une action globale, l'enchaînement des différentes actions est linéaire mais lorsque l'opérateur intervient (action interactive), il peut y avoir une certaine rupture. Dans ce paragraphe nous caractérisons les intentions de l'opérateur, avant d'aborder dans le paragraphe suivant les modifications possible dans le séquencement à travers l'introduction des compatibilités qui définissent la structure dynamique de l'application.

C'est par l'intermédiaire des actions interactives que l'opérateur fournit les données qu'il désire traiter. L'objectif d'une telle action est de spécifier de manière précise cet échange et notamment :

- les informations à fournir à l'opérateur pour l'aider à comprendre la demande (affichages adéquats);
- les aides à fournir à l'opérateur pour répondre correctement;
- la vérification de la validité de la réponse de l'opérateur.

3.3.1. La primitive INTERACTION.

Nous sommes partis de la constatation que le déroulement des actions lors de l'utilisation d'un système (en particulier de C.F.A.O.) était linéaire, sauf lorsque l'opérateur intervenait : dans ce cas, dans la mesure où l'un des objectifs largement répandus est de laisser une grande liberté à l'opérateur, le contexte peut totalement changer. L'opérateur peut avoir des intentions très différentes :

- la désignation ou l'identification. L'opérateur désigne un objet présent dans une scène (ensemble d'objets modélisés);
- la localisation ou la récupération de coordonnées. L'opérateur indique une position sous forme de coordonnées dans une scène;
- la valuation graphonumérique. L'opérateur donne une valeur issue d'une expression dont les opérandes peuvent être numériques, alphanumériques ou graphiques (objets);
- la sélection. L'opérateur montre un menu pour réaliser une action. Nous verrons comment prendre en compte cette intention par l'utilisation des compatibilités dans la définition dynamique;
- l'abandon. L'opérateur abandonne le dialogue en cours.

Dans la plupart des systèmes, et dans GKS en particulier, on définit une primitive de dialogue par intention. Cependant, une telle approche se révèle incompatible avec nos objectifs car elle place l'opérateur dans un contexte figé par la primitive (la récupération d'un scalaire interdit la sélection d'un menu, ...). De plus, l'utilisation d'un tel système n'est pas aisée car il faut bien comprendre la fonction de chaque primitive.

Nous nous sommes donc orientés vers l'utilisation d'une primitive unique de dialogue, appelée INTERACTION dont le rôle est de supporter l'ensemble des intentions de l'opérateur. Chaque interaction sera réalisée par un appel à cette primitive avec les paramètres appropriés. C'est une approche, de plus haut niveau que GKS, qui peut éventuellement servir de couche de base.

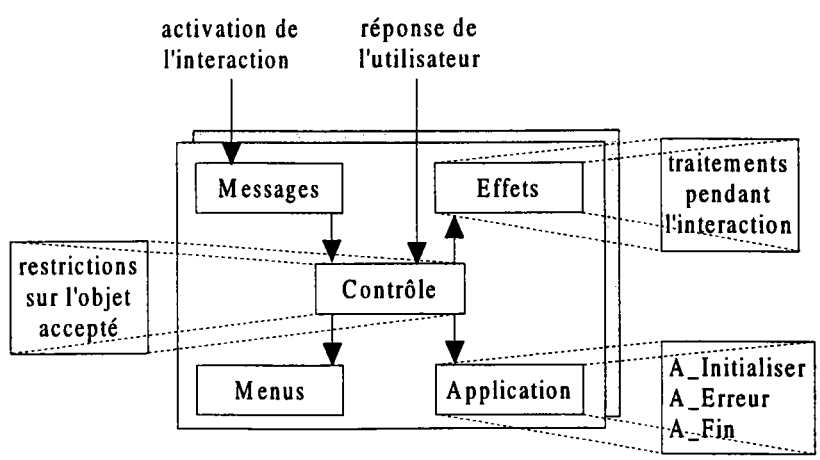


Figure II.5. *Schéma de la primitive INTERACTION.*

Les composants *menus* et *messages* décrivent les informations à fournir à l'opérateur. Le composant *menus* indique les modifications à apporter aux menus et qui peuvent être pris en compte par SACADO. On peut notamment rendre un menu inactif, rendre un domaine actif, ... Le composant *message*s contient des informations à fournir à l'opérateur pour l'aider à comprendre l'action interactive.

Le composant *effets* spécifie l'influence de l'action interactive sur le fonctionnement du système, il décrit le résultat de l'interaction par rapport à l'action de l'opérateur (écho sémantique vers l'opérateur). On y trouve notamment les affichages dynamiques pour aider l'opérateur (affichages temporaires pour aider à comprendre comment le système interprète l'action de l'opérateur). Par exemple, lors de l'interaction de l'extrémité d'un segment, l'effet pourrait être le tracé d'un segment prototype entre le premier point déjà désigné et le pointeur courant.

Le composant *contrôle* définit les restrictions sur l'objet accepté et par conséquent le rôle de l'interaction. Ces restrictions peuvent être exprimées par une liste de types d'objets gérés par le système et par des contraintes que l'objet désigné doit satisfaire afin d'être validé (par exemple, l'opérateur doit désigner un segment de longueur supérieure à 10.0). Ce composant définit le fonctionnement de l'interaction et doit être paramétré convenablement pour toute interaction en fonction des besoins du système à un instant donné. Il reçoit les données de l'opérateur et les traite de façon adéquate.

Le composant *application* décrit l'ensemble des actions à effectuer en fonction de l'avancée de l'interaction (intialisation, ...). On y trouve :

- A_Initialiser, action à exécuter au début de l'interaction;
- A_Fin, action à exécuter à la fin de l'interaction;
- A_Erreur, action à exécuter lorsqu'un objet non valide est désigné par l'opérateur (restrictions ou contraintes sur l'objet non satisfaites).

L'activation d'une interaction entraîne l'exécution de l'action A_Initialiser et la prise en compte des composants *menus* et *messages*. L'interaction attend alors une réponse de l'opérateur. Pendant cette attente, le composant *effets* est actif (il fait écho de l'interprétation du système). Lorsque l'opérateur donne une réponse, l'objet est vérifié par l'interaction en accord avec les contraintes spécifiées dans le composant *contrôle*. Si l'objet est valide, on dit que l'interaction est validée, l'action A_Fin est exécutée et l'exécution continue en accord avec le graphe de l'action globale. Dans le cas contraire, l'action A_Erreur est exécutée et l'interaction attend une nouvelle réponse de l'opérateur.

Avec cette approche, l'exécution est guidée par les données (toute interaction est considérée comme une donnée). À chaque interaction correspond un comportement standard pour y répondre. Par exemple, pour une interaction demandant un segment, la désignation est le comportement standard. Cette primitive unique se rapproche de la notion de caractéristiques : INTERACTION englobe ce que l'on veut obtenir et comment l'obtenir. C'est notre brique de base pour la construction d'un système

interactif de C.F.A.O. Mais pour en augmenter la structuration, nous introduisons les opérateurs de composition.

3.3.2. Les opérateurs de composition.

SACADO fournit le concept des opérateurs de composition pour combiner des interactions et construire par ce biais des interactions composées. Ces opérateurs sont des interactions de plus haut niveau, ils respectent la même décomposition en cinq composants de la primitive INTERACTION pour garantir la cohérence du système. Les trois opérateurs que l'on peut trouver sont ET, OU et ENSEMBLE. Dans les paragraphes suivants, nous utiliserons interaction en lieu et place de la primitive INTERACTION ou d'une interaction composée.

3.3.2.1. L'opérateur ET.

Cet opérateur est défini entre plusieurs interactions : ET ($Inter_1$,.., $Inter_n$). Il est bien adapté lorsque l'opérateur doit répondre à plusieurs interactions alors qu'il n'y a pas d'ordre précis à respecter. Par exemple la construction d'un cercle à partir d'un centre et d'un rayon positif.

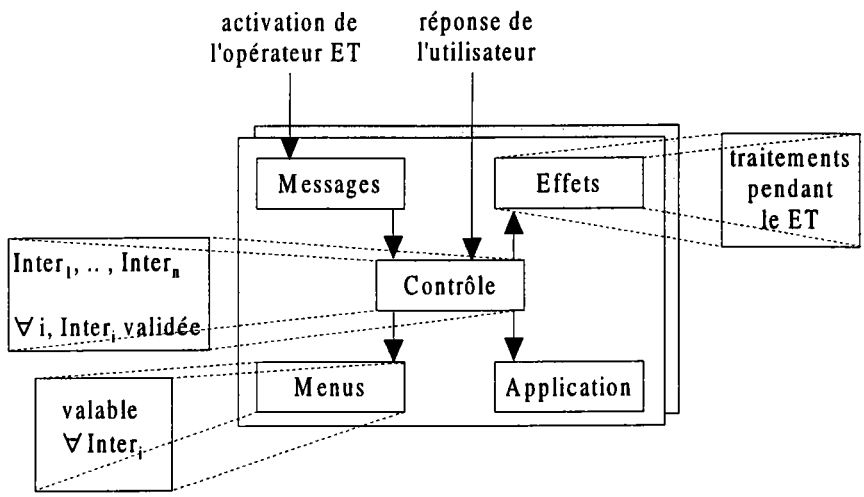


Figure II.6. L'opérateur ET.

Le composant *contrôle* définit toujours le rôle de l'interaction, ici un ET. Il contient donc les différentes interactions $Inter_1$, .., $Inter_n$ ainsi que la condition de validation de cet opérateur : toutes les interactions $Inter_i$ doivent être validées. Un historique est conservé dans ce composant pour déterminer quelles sont les interactions non encore validées (actives).

Les composants *messages*, *menus* et *effets* contiennent des informations qui sont valables pour toutes les interactions définies dans le composant *contrôle* (par exemple un menu inactif pour toutes les interactions $Inter_i$).

L'activation d'un ET entraîne l'activation de toutes les interactions $Inter_i$ qui le composent. Le système attend alors une réponse de l'opérateur. Pendant cette attente, les composants *effets* des interactions non validées sont actifs (le système aide l'opérateur à répondre par un écho sélectif). Lorsque l'opérateur

donne une réponse, l'objet est envoyé à toutes les interactions du ET non encore validées (l'opérateur conserve en interne un historique de validation). Si toutes les interactions $Inter_i$ sont validées, on dit que le ET est validé et l'exécution continue en accord avec le graphe de l'action globale. Dans le cas contraire, le ET attend la réponse suivante de l'opérateur.

3.3.2.2. L'opérateur OU.

Comme l'opérateur précédent, l'opérateur OU s'applique à plusieurs interactions : OU ($Inter_1$, .., $Inter_n$). Il est utilisé lorsque l'opérateur doit répondre à une interaction et que le traitement à faire est différent selon la réponse. Par exemple une action globale dont les traitements diffèrent selon l'objet désigné par l'opérateur : un cercle ou une ellipse.

Le composant *contrôle* contient les différentes interactions $Inter_1$, .., $Inter_n$ ainsi que la condition de validation de cet opérateur : une des interactions $Inter_i$ doit être validée.

Les composants *messages*, *menus* et *effets* contiennent des informations qui sont valables pour toutes les interactions définies dans le composant contrôle (par exemple, un message d'aide valable pour toutes les interactions).

L'activation d'un opérateur OU entraîne l'activation de toutes les interactions $Inter_i$ qui le composent. Le système attend alors une réponse de l'opérateur. Pendant cette attente, les composants *effets* des interactions sont actifs (le système aide l'opérateur à répondre en autorisant tous les échos). Lorsque l'opérateur donne une réponse, l'objet est envoyé à toutes les interactions de l'opérateur OU. Si l'une des interactions $Inter_i$ est validée, on dit que l'opérateur OU est validé et l'exécution continue en accord avec le graphe de l'action globale. Dans le cas contraire, l'opérateur OU attend une autre réponse de l'opérateur.

3.3.2.3. L'opérateur ENSEMBLE.

Cet opérateur de composition est utilisé lorsque l'opérateur doit fournir un ensemble d'objets. Les objets doivent être décrits par l'utilisation d'une interaction Inter : ENSEMBLE (Inter).

Le composant *contrôle* contient l'action interactive Inter ainsi que les conditions que doit vérifier l'ensemble construit pour être validé (ces conditions se rapprochent des restrictions introduites pour la primitive INTERACTION). Par exemple, le cardinal de l'ensemble des objets doit être égal à 10.

Les composants *messages*, *menus* et *effets* contiennent des informations qui sont valables pour l'interaction Inter définie dans le composant contrôle.

L'activation d'un opérateur de composition ENSEMBLE entraîne l'activation de l'interaction Inter. Lorsque l'opérateur donne une réponse, l'objet est envoyé à l'interaction Inter. Si elle est validée, l'objet est ajouté à l'ensemble. Cet ensemble est alors vérifié en accord avec les contraintes spécifiées dans le composant *contrôle*. Si l'ensemble est valide, on dit que ENSEMBLE est validé et l'exécution continue

en accord avec le graphe de l'action globale. Dans le cas contraire, ENSEMBLE attend une nouvelle réponse de l'opérateur.

4. LA DÉFINITION DYNAMIQUE.

La définition statique précédente nous indique que le contexte du système peut être défini par ContexteMenus, le contexte des menus, et ContexteExécution, le contexte d'exécution :

Contexte = (ContexteMenus, ContexteExécution)

Le contexte d'exécution est représenté par le couple (Action, Interaction)

avec Action = action globale en cours d'exécution (Action $\in$ ACTIONS GLOBALES).
Interaction = état de l'action globale représenté par l'interaction courante (le dialogue en cours).

Ce contexte montre que l'opérateur ne peut avoir qu'une seule action globale en cours à un instant donné. Mais généralement, une tâche peut être considérée comme une suite de sous-tâches planifiées hiérarchiquement conformément au modèle de l'interaction homme-machine. Cette hiérarchie n'est souvent formulée qu'à l'exécution lorsque l'opérateur se rend compte qu'une sous-tâche est nécessaire pour pouvoir accomplir la tâche initiale [CHA 93].

Un système doit donc permettre à l'opérateur de suivre sa propre logique de décomposition tout en respectant les impératifs du logiciel. Cette possibilité constitue un des problèmes majeurs du domaine des interfaces homme-machine. Il s'agit de réduire la distance d'exécution, rapprocher ce que veut faire l'opérateur du "comment le faire" à l'aide du système.

Dans SACADO, une action globale peut seulement être interrompue lors de l'exécution d'une interaction. En effet, lors d'une phase de dialogue, l'opérateur est sollicité afin de fournir les informations nécessaires au traitement d'une fonctionnalité. À cet instant, il est possible de modéliser les comportements possibles de l'opérateur et par conséquent, ce que doit faire le dialogue face à l'opérateur.

Dans cette situation, on définit le comportement standard de l'opérateur comme étant la réponse directe à une interaction. Mais, il peut également vouloir réaliser une autre action à travers la sélection d'un menu (l'exécution d'une autre fonctionnalité). On définit alors le terme de compatibilité par la réaction qu'aura le dialogue lors de l'interruption d'une interaction par un menu.

Dans les paragraphes qui suivent, nous définissons précisément les compatibilités intégrées dans le système SACADO. Nous indiquons comment les représenter pour une application donnée et surtout les modifications que cela entraîne aussi bien pour le programmeur d'interfaces que pour l'opérateur dans le développement et l'utilisation de l'interface utilisateur de SACADO.

4.1. Les compatibilités.

À un instant donné, le système se trouve dans un certain contexte qui est simplement défini par les menus sélectionnés depuis la racine par l'opérateur. Si lors d'une phase de dialogue, l'opérateur décide de sélectionner un nouveau menu (par conséquent, une nouvelle action), le contexte peut totalement changer.

Nous introduisons des concepts se rapprochant des outils dont dispose un programmeur : fonctions, procédures, données ... Ces concepts traduisent les différentes possibilités offertes à un opérateur pour répondre à une interaction; en effet, il se trouve confronté à plusieurs alternatives :

- répondre directement par un objet valide avec les données dont il dispose; c'est le comportement standard;
- répondre indirectement, en utilisant une possibilité offerte par le système; nous parlons de compatibilité *locale*. L'opérateur construit ou recherche l'objet demandé par l'interaction en exécutant une nouvelle action (construction d'un objet oublié ...). Il s'agit de préciser un contexte par le choix d'un menu qui n'a de sens que par rapport au contexte courant. L'action associée est exécutée immédiatement et, lorsqu'elle termine, le contexte suspendu reprend au point d'interruption avec le ou les objet(s) résultat(s);

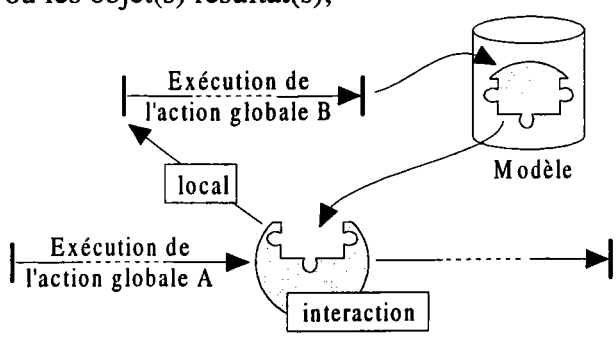


Figure II.7. *Compatibilité locale.*

- répondre directement après avoir modifié l'environnement; cette compatibilité est appelée *immédiate*. Cela consiste à exécuter une nouvelle action pour faciliter la réponse (déplacer un objet pour rendre visible l'objet avec lequel l'opérateur veut travailler ...). L'opérateur entre dans un nouveau contexte et lorsque l'action termine, il retrouve le contexte suspendu;

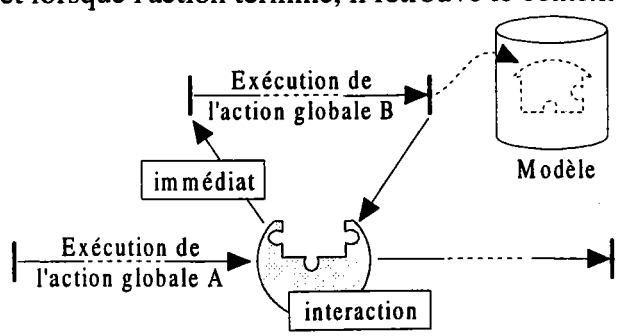


Figure II.8. *Compatibilité immédiate.*

- abandonner l'interaction; cette compatibilité est appelée *différée*. Cela consiste à exécuter une action incompatible avec l'attente du système, tout en garantissant d'annuler (ou de terminer) l'action en cours dans de bonnes conditions (destruction d'un objet pendant la construction d'un contour ...). On dit que l'action est différée car son exécution est retardée par la recherche d'un contexte compatible.

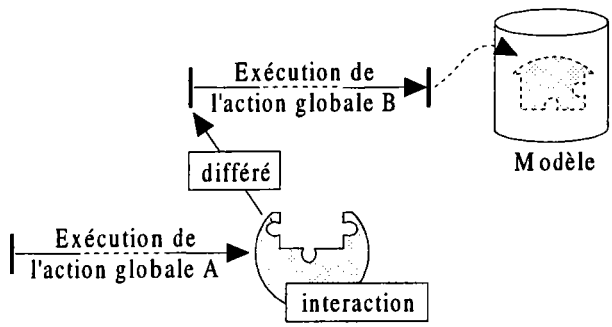


Figure II.9. *Compatibilité différée.*

Une première implantation utilisant des menus locaux de type point et basée sur une primitive INTERACTION intégrée à la partie applicative a permis de confirmer le bien fondé de ces concepts [TOT 89]. Dans les paragraphes qui suivent, nous nous attachons à une spécification précise des compatibilités afin de généraliser ces concepts et d'obtenir une implantation réelle de SACADO s'appuyant notamment sur la définition dynamique de l'architecture et la séparation dialogue-application.

Les compatibilités définissent une organisation sémantique des menus transversale par rapport à la hiérarchie. Cette définition des compatibilités est une solution originale à l'explosion combinatoire des réseaux de transitions (une simple compatibilité offre un chemin supplémentaire vers une autre action globale) et au contrôle des fils d'activités multiples que l'on trouve dans les applications récentes. Par ce biais, le dialogue indique clairement les menus (actions globales) autorisés en accord avec le système. L'opérateur peut ainsi se situer et se repérer par rapport à sa tâche. Une condition d'application de ces compatibilités est qu'il ne peut exister deux liens de compatibilités directs différents pour un même menu. Par exemple, un menu ne peut être à la fois local et immédiat pour une même interaction car le système ne saurait quelle compatibilité choisir.

Bien entendu, il est impératif de représenter le plus fidèlement possible l'état de l'application pour informer l'opérateur du contexte dans lequel il se trouve (couleurs, styles, aides, ...). Mais avec les compatibilités, la définition du contexte d'exécution change totalement. Il faut maintenant tenir compte des fils d'activités multiples. Chaque fil d'activité est représenté par un triplet (Action, Inter, Comp) définissant son rôle :

- Action, action globale exécutée (Action $\in$ ACTIONS GLOBALES);
- Inter, état de l'action globale Action représenté par l'interaction suspendue;
- Comp, compatibilité du menu ayant déclenché l'activation (immédiat ou local).

Le contexte d'exécution est alors représenté par une suite de triplets représentant les fils d'activité dans l'ordre chronologique de leur lancement :

$$\text{ContexteExécution} = \{ (Act_1, Inter_1, Comp_1) \ldots (Act_n, Inter_n, Comp_n) \}.$$

Nous définissons quelques termes importants :

- l'action globale Act_1 est appelée action globale principale. C'est l'action globale associée au premier menu terminal sélectionné par l'opérateur;
- l'action globale Act_n est l'action globale courante et $Inter_n$ est l'interaction courante. C'est l'action globale associée au dernier menu terminal sélectionné par l'opérateur.

Le contexte des menus que l'on note ContexteMenus est maintenant obtenu par application successive des compatibilités associées aux actions globales Act_i par rapport à un contexte de départ que l'on appelle ContexteDépart :

ContexteDépart = { (menu, état) | menu $\in$ MENU et état $\in$ (inactif, immédiat, local, différé) }
et $\forall$ menu $\in$ MENUS, $\exists$ (menu, état) $\in$ ContexteDépart

ContexteMenus = f (ContexteDépart , ContexteExécution)
et f la fonction d'application des compatibilités que l'on définit ultérieurement.

On obtient donc le contexte de l'application par :

Contexte = (ContexteExécution, ContexteMenus)

La définition même des action globales et des compatibilités garantit l'indépendance des actions globales entre elles. Il n'est pas nécessaire, et certainement pas souhaitable, qu'une action globale locale ou immédiate "connaisse" l'interaction dont elle est issue. Cette indépendance permet un développement rapide d'un grand nombre d'actions globales par des programmeurs différents, le lien entre elles se faisant grâce aux compatibilités. Dans les paragraphes qui suivent, nous précisons comment les compatibilités sont représentées par rapport à la définition statique de l'application et comment les compatibilités évoluent en fonction du contexte.

4.2. La définition des compatibilités.

Une compatibilité n'est définie (n'a de sens) que lorsque l'opérateur interagit avec le système à travers une ou plusieurs interactions. C'est à cet instant qu'elle prend toute sa signification en influençant et en orientant la réponse de l'opérateur. La spécification de ces compatibilités peut se faire principalement de deux façons :

- au niveau des menus;
- au niveau des interactions elles-mêmes.

Lorsque le programmeur d'interfaces définit des compatibilités entre des menus, les compatibilités se propagent par héritage vers les descendants de la hiérarchie (un menu fils reprend les compatibilités de son père) en respectant certaines règles :

- sauf indication contraire, un menu fils hérite des compatibilités du père. Définir un menu ZOOM immédiat du menu SEGMENT revient à le définir immédiat de toutes les constructions qui sont fils de SEGMENT (segment par deux points, segment tangent à deux cercles, ...);
- sauf indication contraire, une action globale hérite des compatibilités associées au menu terminal correspondant. On dit que les compatibilités du menu terminal forment les compatibilités de départ de l'action globale. On note COMPATIBILITES (Action) les compatibilités de départ de l'action globale Action;

Rappelons que les compatibilités n'ont de sens que dans les phases de dialogue des actions globales associées. Cela signifie que les compatibilités définies pour une action globale sont celles définies pour toutes les interactions de cette action. Cet héritage implique bien sûr que les menus que l'on rend compatibles soient cohérents avec l'interaction concernée. Ceci est particulièrement vrai pour le lien local. Rendre un menu local à un autre menu, entraîne par l'héritage que le menu devient local à toutes les interactions de l'action globale associée. Il est donc impératif, pour garantir la validité du dialogue, que le menu local et l'interaction soient cohérents par rapport aux objets demandés.

Le programmeur d'interfaces peut également définir les compatibilités directement pour chaque interaction. Le composant *menus* est étendu pour contenir ces compatibilités. Cette association permet notamment d'apporter des précisions sur les compatibilités héritées : par exemple, le menu ZOOM est déclaré inactif pour une interaction donnée alors qu'il est immédiat pour l'action globale. Les compatibilités d'une interaction dépendent de sa nature (primitive INTERACTION ou opérateur de composition). Soit COMPATIBILITE, la fonction donnant les compatibilités d'une interaction.

Pour la primitive INTERACTION, le système autorise simplement tous les fils d'activités (compatibilités) définis pour l'interaction concernée.

COMPATIBILITE (INTERACTION) = *menus*

Pour l'opérateur de composition ET, le système autorise les fils d'activités du ET (ce sont les compatibilités destinées à guider l'opérateur pendant toute l'interaction composée) et les fils d'activités des interactions auxquelles l'opérateur doit encore répondre.

$$\text{COMPATIBILITE (ET)} = \textit{menus} \quad + \sum_{i=1}^{n} \{\text{COMPATIBILITE (Inter}_i\text{)} \mid \text{Inter}_i \text{ non validée}\}$$

De cette manière, l'interaction fait évoluer les compatibilités au fur et à mesure des réponses de l'opérateur puisque COMPATIBILITE (ET) change à chaque réponse valide de l'opérateur. Lorsque le

système demande un point et un scalaire toutes les compatibilités sont utilisées mais lorsque l'opérateur a déjà répondu par un point, seules les compatibilités du scalaire sont utilisées.

Pour l'opérateur de composition OU, le système autorise les fils d'activités du OU (ce sont les compatibilités destinées à guider l'opérateur pendant toute l'interaction composée) et les fils d'activités des interactions qui le composent.

$$\text{COMPATIBILITE (OU)} = \textit{menus} \quad + \sum_{i=1}^{n} \text{COMPATIBILITE (Inter}_i\text{)}$$

Pour l'opérateur de composition ENSEMBLE, le système autorise les fils d'activités de ENSEMBLE (ce sont les compatibilités destinées à guider l'opérateur pendant toute l'interaction composée) et les fils d'activités de l'interaction qui le compose.

$$\text{COMPATIBILITE (ENSEMBLE)} = \textit{menus} + \text{COMPATIBILITE (Inter)}$$

Comme l'indique les définitions précédentes, sauf indication contraire, les opérateurs de composition héritent des compatibilités des interactions qui le composent (on peut également le voir dans le sens inverse). Cela augmente encore la dynamicité du système face aux modifications des compatibilités. On obtient une hiérarchie de compatibilités, hiérarchie représentée par les opérateurs de composition.

4.3. La représentation des compatibilités.

Les compatibilités permettent de créer des dialogues contextuels qui répondent aux besoins ponctuels de l'opérateur. Pour une interaction donnée, les compatibilités offrent des possibilités à l'opérateur pour continuer sa réflexion. Considérons une application composée de trois menus principaux : POINT, CERCLE et SEGMENT. Le menu POINT est non terminal et possède deux menus fils terminaux : COORDONNEES (construction d'un point à partir de coordonnées) et INTERSECTION (construction d'un point par intersection de deux objets). Les menus CERCLE et SEGMENT sont terminaux.

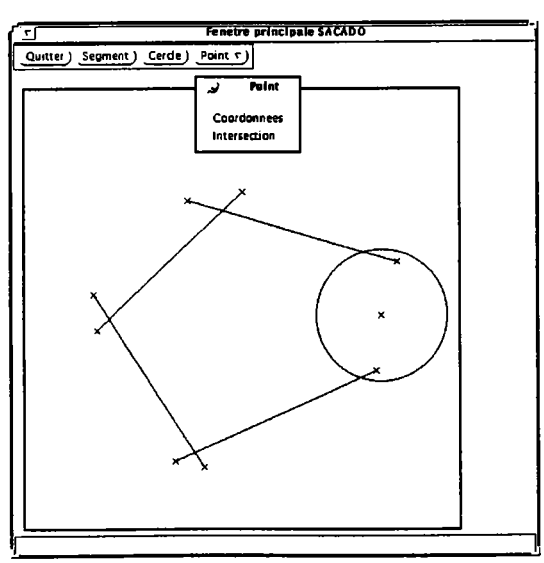


Figure II.10. *Exemple d'application SACADO.*

La représentation des compatibilités peut se faire selon deux approches :

- une description indépendante du contexte (par des graphes);
- une description en fonction du contexte (par des arbres).

4.3.1. Les graphes de compatibilité.

Cette première approche demande au programmeur d'interfaces de décrire les compatibilités pour chaque menu et chaque interaction de l'application (des outils doivent bien sûr aider à cette description et notamment des bibliothèques dont nous reparlerons dans le chapitre 3). Cette description se fait donc pour chaque entité indépendamment des autres et surtout, sans tenir compte du chemin utilisé par l'opérateur pour l'atteindre. On peut dire qu'elle s'apparente à la définition d'un graphe de compatibilités.

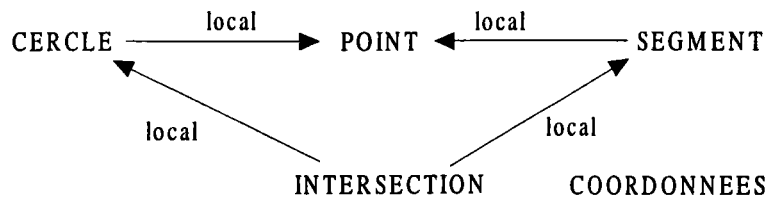


Figure II.11. *Représentation indépendante du contexte.*

Cette représentation implique que SEGMENT aura toujours pour local le menu POINT qu'il s'agisse de la construction d'un segment en tant qu'action principale ou de la construction d'un segment en tant que local du menu INTERSECTION. Une telle approche peut aboutir parfois à des situations pardoxales notamment par transitivité (en particulier des boucles entre menus).

4.3.2. Les arbres de compatibilité.

Cette seconde approche s'appuie sur la notion de contexte. Le programmeur d'interfaces autorise les compatibilités en fonction du contexte d'exécution, il doit décrire tous les chemins qu'il désire autoriser à travers la définition des compatibilités.

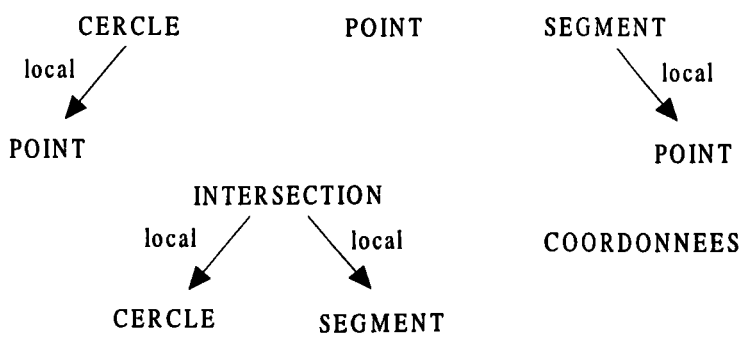


Figure II.12. *Représentation en fonction du contexte.*

Cette représentation permet de préciser les compatibilités suivant le contexte et en particulier, en fonction de l'action globale principale. Dans l'exemple, le menu POINT est local au menu SEGMENT

lorsque l'action globale associée à SEGMENT est principale mais pas lorsque cette même action est sélectionnée en tant que local du menu INTERSECTION. La représentation s'apparente alors à des arbres de compatibilités (un arbre par compatibilité).

4.3.3. Synthèse.

Les deux approches possèdent chacune leurs avantages et leurs inconvénients. La représentation par graphe évite toute duplication de menus dans la définition des compatibilités. Le programmeur d'interfaces indique uniquement les liens invariants mais n'a aucune influence sur les situations paradoxales qui doivent donc lui être signalées par les outils de description. Dans ce cas, la solution se trouve dans la représentation par arbres qui autorise la description explicite des compatibilités dans le contexte posant problème. Cependant, son utilisation dans un cadre plus large impose un travail trop important au programmeur d'interfaces par la multiplication des chemins en fonction du nombre de menus et d'interactions.

La solution est donc naturellement de se servir des deux approches en utilisant les avantages de chacune d'elle : la dynamicité du graphe (compatibilités implicites et le système évolue tout seul) et l'adaptation contextuelle des arbres (compatibilités explicites en cas de problèmes). Nous considérons une compatibilité comme un ensemble de couples :

$$\text{Compatibilité} = \{ (\text{menu}, \text{état}) \mid \text{menu} \in \text{MENU et état} \in (\text{inactif, immédiat, local, différé}) \}$$

Tous les menus de l'application n'ont pas à apparaître dans une compatibilité, seuls les menus dont le programmeur d'interfaces désire modifier la compatibilité sont à définir. Pour tout couple (menu, état) d'une compatibilité, cela signifie que le programmeur d'interfaces veut appliquer la compatibilité état à menu.

Le composant *menus* des interactions contient donc une compatibilité telle qu'elle est définie ci-dessus par un ensemble de couples. L'opérateur + que l'on a utilisé dans le paragraphe précédent pour construire les compatibilités d'un opérateur de composition (COMPATIBILITE (op)) est simplement une union des compatibilités. Lorsque des ambiguïtés sont détectées (deux compatibilités différentes pour un même menu), un outil doit en informer le programmeur d'interfaces. On peut également fixer des priorités, par exemple, on peut donner la priorité aux compatibilités des opérateurs de composition par rapport aux compatibilités des interactions qui les composent.

Il est également nécessaire de préciser le composant *application* en introduisant trois nouvelles actions destinées à la gestion des compatibilités :

- A_Différé, action à exécuter dans le cas d'un appel de menu différé afin de quitter l'action interactive dans de bonnes conditions;
- A_Local, action à exécuter dans le cas d'un appel de menu local (passage de paramètres au local, ...);

- A_Immédiat, action à exécuter dans le cas d'un appel de menu immédiat (mise à jour de variables locales ...).

Avec les définitions et les choix effectués précédemment, nous possédons une définition précise des compatibilités. Nous nous intéressons dans les paragraphes suivants aux règles d'application des compatibilités dans le but de minimiser le nombre de situations paradoxales et d'éviter au maximum le recours à une description explicite des compatibilités par des arbres.

4.4. L'évaluation des compatibilités.

Le contexte du système est représenté par le contexte d'exécution des actions globales (fils d'activités utilisés) et par le contexte des menus :

$$\text{Contexte} = (\text{ContexteExécution}, \text{ContexteMenus})$$

avec
$$\text{ContexteExécution} = \{ (Act_1, Inter_1, Comp_1) \ldots (Act_n, Inter_n, Comp_n) \}.$$
$$\text{ContexteMenus} = f(\text{ContexteDépart}, \text{ContexteExécution})$$

En partant de ce contexte, il s'agit de préciser le caractère dynamique du système en définissant totalement et précisément la fonction f d'évaluation des compatibilités. Pour cela, nous étudions l'évolution du contexte des menus dans toutes les situations où les compatibilités sont amenées à changer :

- sélection d'un menu compatible (local ou immédiat);
- validation d'une interaction.

4.4.1. Sélection d'un menu compatible.

Lorsque l'opérateur sélectionne un menu terminal associé à une action globale Act_{n+1} compatible (local ou immédiat), le contexte d'exécution change :

$$\text{ContexteExécution} = \{ (Act_1, Inter_1, Comp_1) \ldots (Act_n, Inter_n, Comp_n) (Act_{n+1}, Inter_{n+1}, Comp_{n+1}) \}$$

Cette situation n'est valable que si la nouvelle action globale Act_{n+1} comporte des interactions sinon elle se termine immédiatement et aucune opération n'est nécessaire, exceptée si elle répond à l'interaction. Dans ce cas, il s'agit d'une validation, cas qui est traité dans le paragraphe suivant. Dans notre cas, nous avons supposé que Act_{n+1} comporte au moins une interaction et que l'interaction $Inter_{n+1}$ représente son état (interaction active). Il est donc nécessaire de prendre en compte les nouvelles compatibilités que l'on combine avec les compatibilités courantes. On obtient le nouvel état grâce à l'opération :

$$\text{ContexteMenus} = \text{ContexteMenus} \cup \text{COMPATIBILITE}(Inter_{n+1})$$

Nous définissons l'opérateur $\cup$ utilisé ici pour représenter l'application d'une compatibilité sur un état des menus :

$$\begin{array}{rccccc} \cup : & \text{ContexteMenus} & \text{x} & \text{Compatibilité} & \longrightarrow & \text{ContexteMenus} \\ & (\text{Contexte} & , & \text{Comp}) & \longmapsto & \text{ContexteRésultat} \end{array}$$

Le nouvel état de chacun des menus est obtenu par application de la nouvelle compatibilité demandée. Ainsi, pour tout menu m tel que $(m,e) \in$ Contexte :

- si $\exists (m, e') \in$ Comp alors $(m, \text{transformation } (e, e')) \in$ ContexteRésultat
- dans le cas contraire, l'état du menu est inchangé : $(m, e) \in$ ContexteRésultat

Cette première définition montre que l'état d'un menu dépend de son état courant et de l'état qui est demandé. Cette dépendance permet d'éviter au maximum les situations paradoxales évoquées au paragraphe précédent. La table ci-dessous donne, pour un menu donné, un exemple de règles de transformation des compatibilités :

Avant / Requis	Inactif	Immédiat	Local	Différé
Inactif	*Inactif*	*Inactif*	*Inactif*	*Différé*
Immédiat	*Immédiat*	*Immédiat*	*Immédiat*	*Différé*
Local	*Local*	*Local*	*Local*	*Différé*
Différé	*Différé*	*Différé*	*Différé*	*Différé*

La définition précédente est insuffisante car l'opérateur $\cup$ ne s'applique qu'aux menus qui apparaissent dans la compatibilité or, il faut également modifier certains menus dont l'état devient incohérent par rapport au nouveau contexte d'exécution. Par exemple, un menu local ne reste pas local si on ne le demande pas explicitement ou si le type de sa réponse n'est plus compatible. L'opérateur $\cup$ ne s'applique donc pas seulement aux menus spécifiés mais également à ceux qui ne sont pas précisés. La table ci-dessous donne un exemple de transformations :

Avant / Requis	Inactif	Immédiat	Local	Différé
Aucun	*Inactif*	*Immédiat* ou *Différé*	*Différé*	*Différé*

Ces tables sont très importantes. Elles déterminent la dynamique de l'interface, son évolution par rapport aux actions de l'opérateur en fonction des spécifications du programmeur d'interfaces (évolution du contexte en fonction des actions de l'opérateur). Il est essentiel de bien déterminer les changements à prendre en compte. Tout changement dans ces tables peut modifier radicalement le comportement de l'interface.

On peut prendre comme exemple deux menus M1 et M2 locaux d'une interaction Inter. Lorsque l'opérateur sélectionne le menu M1, l'action globale associée est exécutée et les compatibilités modifiées en accord avec les règles édictées précédemment. Dans la mesure où M2 n'apparaît pas dans ces compatibilités, la deuxième table est utilisée et M2 devient différé pour la première interaction Inter' de l'action globale associée à M1.

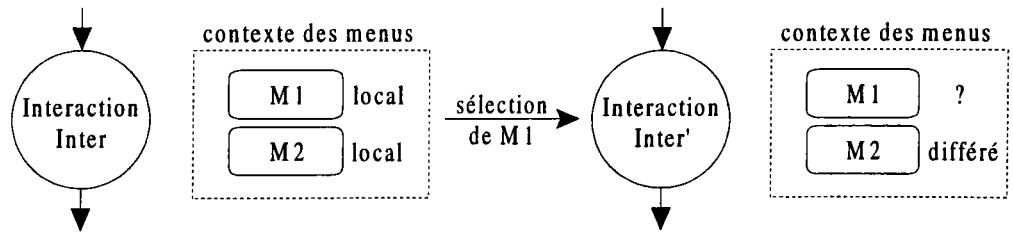


Figure II.13. *Évolution d'une compatibilité locale.*

Cette modification est tout à fait cohérente avec la définition d'un différé car la sélection de M2 comme différé entraînera l'annulation totale de l'action globale associée à M1 jusqu'à l'exécution de M2 en tant que local de Inter (c'est le premier contexte dans lequel M2 n'est plus différé mais local). Cependant, jusqu'à présent un différé était considéré comme "incompatible". En fait, dans beaucoup de cas, après application des transformations, il s'agit plutôt d'un effet "autre choix". Lorsque M2 devient différé, il possède un aspect plutôt déroutant pour un opérateur. Il est souhaitable qu'il apparaisse avec un aspect indiquant clairement que ce menu est une alternative, un autre choix à celui qui est sélectionné et non une incompatibilité.

Pour tenir compte de ces changements, il faut dissocier l'influence d'un menu qui est défini par sa compatibilité et son aspect. Un menu différé peut avoir plusieurs aspects (ou explications fournies à l'opérateur) alors qu'il a toujours la même définition interne au dialogue. Les deux tables que nous avons présentées sont des exemples de ce que l'on peut obtenir. Nous envisageons des tables qui tiennent compte de la compatibilité de l'action appelée (immédiat ou local) ou du contexte d'exécution. Nous obtiendrons alors plus de précisions sur les changements de compatibilités des menus.

En conclusion, on obtient finalement le contexte des menus par :

$$\begin{aligned} \text{ContexteExécution} &= \{ (Act_1, Inter_1, Comp_1) \ldots (Act_n, Inter_n, Comp_n) \}. \\ \text{ContexteMenus} &= \text{ContexteDépart} \cup \text{COMPATIBILITE}(Inter_1) \\ &\quad \ldots \\ &\quad \cup \text{COMPATIBILITE}(Inter_n) \end{aligned}$$

4.4.2. Validation d'une interaction.

Lorsque l'opérateur répond à l'interaction courante par un objet valide, les compatibilités sont modifiées. Le contexte d'exécution peut totalement changer, on passe au contexte ContexteExécution , à savoir de l'interaction Inter à l'interaction Inter' :

$$\text{ContexteExécution} = \{ (Act_1, Inter_1, Comp_1) \ldots (Act_n, Inter'_n, Comp_n) \}$$

Nous avons supposé que l'action Act_n ne prenait pas fin après la réponse de l'opérateur. Dans le cas contraire, les modifications sont indiquées à la fin de ce paragraphe. Nous décrivons comment déterminer le nouveau contexte des menus à partir de l'ancien (ContexteMenus)..

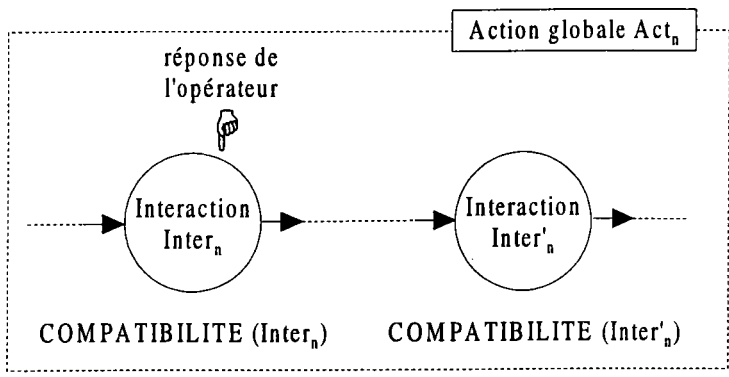


Figure II.14. *Validation d'une interaction.*

Dans un premier temps, il est nécessaire de retirer les compatibilités mises en place par l'interaction $Inter_n$ pour retrouver le contexte de menus dans son état initial. On obtient cet état grâce à l'opération suivante :

$$\text{ContexteMenus} = \text{ContexteMenus} \cup^{-1} \text{COMPATIBILITE}(Inter_n)$$

L'opérateur $\cup^{-1}$ est simplement l'application inverse d'une compatibilité. On peut représenter cet opérateur par une compatibilité qui contient pour chacun des menus leur ancienne compatibilité, la compatibilité inverse étant construite lors de l'application de l'opérateur $\cup$.

Après activation de l'interaction $Inter'_n$, on obtient finalement :

$$\begin{aligned} \text{ContexteMenus} = \quad & (\text{ContexteMenus} \cup^{-1} \text{COMPATIBILITE}(Inter_n)) \\ \cup \quad & \text{COMPATIBILITE}(Inter'_n) \end{aligned}$$

Au cas où l'action Act_n prend fin avec l'interaction $Inter_n$, les compatibilités doivent simplement être remises dans leur état initial par :

$$\text{ContexteMenus} = \text{ContexteMenus} \cup^{-1} \text{COMPATIBILITE}(Inter_n)$$

Si Act_n était une action globale locale à l'interaction $Inter_{n-1}$ ($Comp_n$ = local), on applique le même raisonnement de validation à cette interaction avec le résultat de l'action globale Act_n. Par contre, si Act_n était une action globale immédiate à $Inter_{n-1}$ ($Comp_n$ = immédiat), le contexte des menus est déjà restauré et les dialogues peuvent continuer sans autre modification.

4.5. Les compatibilités explicites.

Avec la définition des compatibilités donnée ci-dessus, il est impossible d'utiliser un menu avec une autre compatibilité que celle déclarée. On doit pouvoir tracer un segment en immédiat alors que le menu est local. Dans le cas contraire, l'opérateur est obligé d'inverser son raisonnement et d'effectuer son action en s'adaptant aux compatibilités données par le système, ce qui va à l'encontre du caractère opportuniste de l'opérateur. Ce cas de figure est encore plus flagrant avec les différés introduits par les tables de conversion. Lorsqu'un menu devient différé par une table, l'opérateur doit pouvoir l'utiliser avec une autre compatibilité s'il le désire. Il est donc important d'introduire la notion de compatibilité explicite où l'opérateur choisit la compatibilité avec toutefois une vérification de la cohérence (un différé "incompatible" ne peut être immédiat). Dans ce sens, nous envisageons de déclarer des menus pouvant posséder plusieurs compatibilités (conservation d'un historique des compatibilités pour vérifier la cohérence de l'interface).

4.6. Les effets et les compatibilités.

L'indépendance entre les actions globales est une des particularités de SACADO. Chaque action globale est décrite indépendamment des autres. Généralement, une action globale locale ou immédiate ne connaît pas l'action qui a permis son exécution. Par exemple, l'action de création d'un point ne doit pas faire la différence entre la création d'un point comme action globale principale ou comme locale de l'extrémité d'un segment. Pourtant, le contexte est totalement différent et le système doit contrôler la continuité des échos spécifiés dans le composant *effets* des interactions. Par suite, l'héritage des effets est un problème crucial. Dans l'exemple précédent, le fait de sélectionner le menu POINT comme local de l'interaction ne doit pas stopper l'effet de l'interaction, bien au contraire. Tout au long de l'exécution du local, l'effet de l'interaction interrompue doit continuer en fonction de l'avancée du local pour continuer à informer l'opérateur. Ainsi, lorsqu'un menu local est sélectionné par l'opérateur, les effets de l'interaction interrompue restent actifs.

4.7. Le différé et l'annulation.

Lorsque l'on parle d'annulation, on désire le plus souvent revenir à l'interaction précédente. Ainsi, l'annulation revient à parcourir le chemin inverse que l'on a emprunté dans l'action globale en annulant les différentes actions. Dans le cas du différé, il s'agit du même problème : il faut rechercher un contexte de menus pour lequel le menu n'est pas différé. On peut procéder par annulations successives jusqu'à trouver un ContexteMenus pour lequel le menu n'est plus différé.

Se pose donc le problème de la destruction des différents objets créés par les interactions ou les actions que l'on désire annuler. Les actions non interactives ne peuvent être annulées que par le programmeur d'applications, si elles manipulent des structures de données qui lui sont propres. Dans le cas où ces actions ne s'appuieraient que sur des objets provenant du dialogue, leur annulation peut être prise en compte.

Le problème des interactions est tout autre. Si le résultat d'une interaction est un objet qui existait déjà (objet obtenu par désignation) alors l'annulation revient à une simple réactivation de l'interaction. Par contre, si le résultat de l'interaction est un objet qui provient d'un menu local alors plusieurs solutions sont envisageables :

- conserver le nouvel objet et réactiver l'interaction; la destruction ultérieure des objets est laissée à la charge de l'opérateur;
- détruire le nouvel objet et réactiver l'interaction; seul l'objet ayant répondu à l'interaction est détruit sans tenir compte des éventuels objets ayant servis à sa construction;
- remonter aux actions locales exécutées pour les annuler individuellement.

Le schéma ci-dessous illustre les deux dernières propositions.

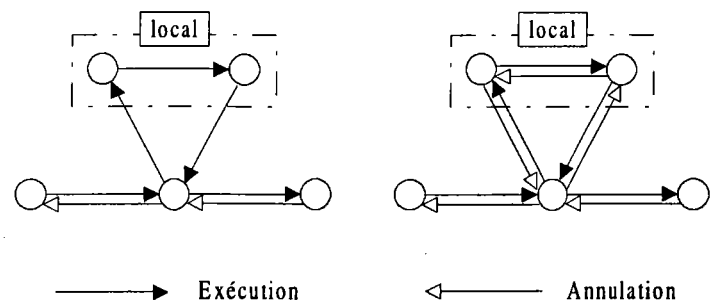
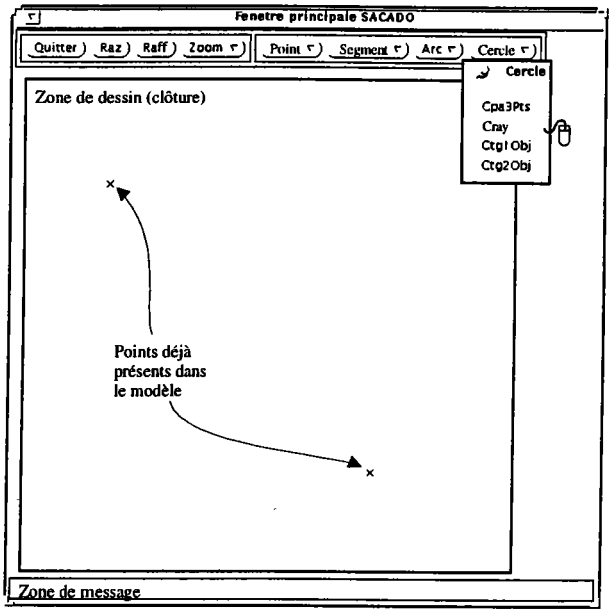


Figure II.15. *L'annulation.*

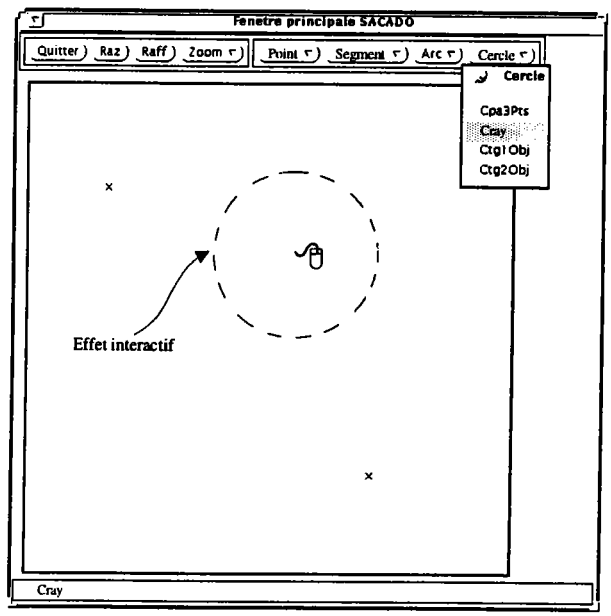
Dans le cas d'une annulation incluant les locaux et les immédiats, un certain nombre de contraintes sont nécessaires. Pour chaque action globale, un historique doit être conservé tant que les actions qui l'ont utilisée ne sont pas terminées. Dans le cas contraire, le système ne pourrait pas retrouver les anciennes valeurs des objets. Cette solution implique un fonctionnement assez lourd du système sans aucune garantie de ne pas dérouter l'opérateur, qui n'est pas censé se souvenir de la manière dont avait été obtenu tel ou tel objet. La solution que nous retenons consiste à ne pas tenir compte des appels locaux ou immédiats et à annuler uniquement les interactions de l'action globale courante. Dans toutes les maquettes réalisées, cette méthode donne entière satisfaction.

5. IMPLANTATION ET EXEMPLE.

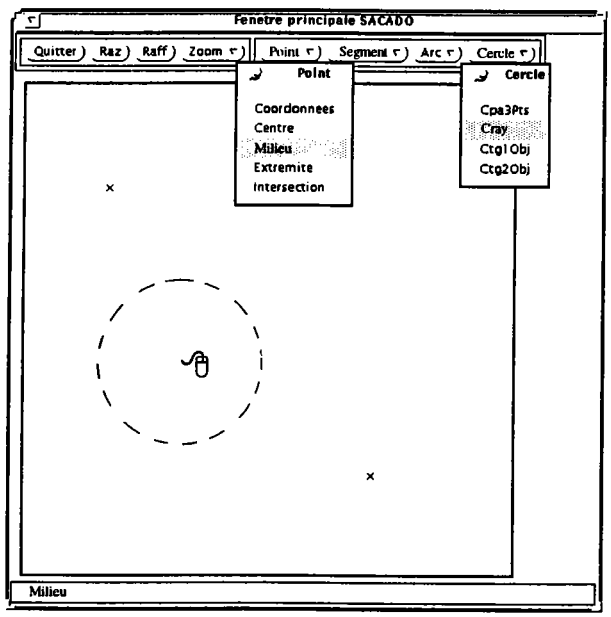
La spécification précise des concepts de SACADO nous a permis de réaliser une nouvelle implantation de ce système sur des stations de travail SUN. Nous avons utilisé OLIT (Open Look Intrinsics Toolbox), une boîte à outils propre à SUN, qui fournit des widgets décrivant les éléments graphiques de l'interface. Ce développement, réalisé en C++ (langage orienté objets), a été effectué en collaboration avec Tony PIPIERI dans le cadre d'une thèse d'ingénieur C.N.A.M. [PIP 95] (cf. annexe A). Pour illustrer les concepts décrits dans ce chapitre, nous présentons la création d'un cercle de rayon 300.0 dont le centre doit être le milieu d'un segment non encore créé reliant 2 points pré-existants.



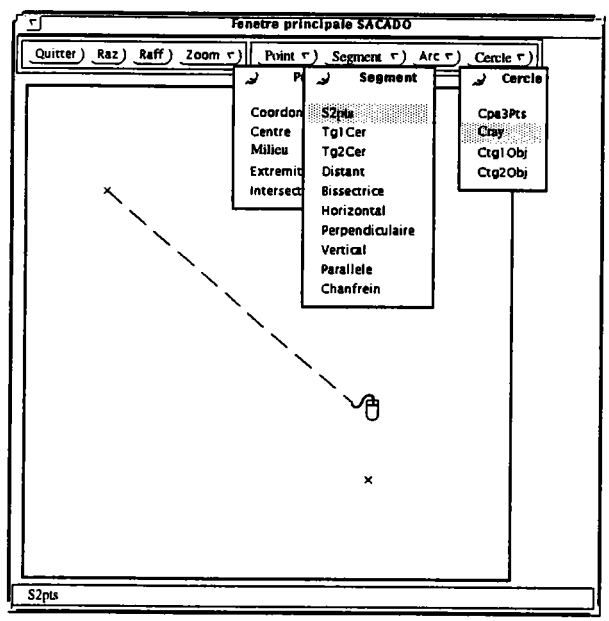
L'opérateur désire construire le cercle et doit donc sélectionner le menu terminal CRAY de construction d'un cercle à partir d'un centre et d'un rayon.



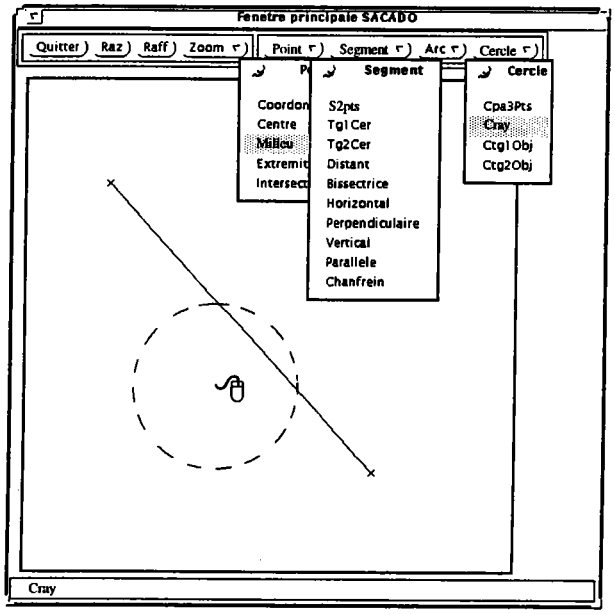
Après avoir donné la valeur du rayon, l'opérateur doit indiquer le centre du cercle. Il doit désigner un point ou le créer par un local (ici POINT et ses fils sont locaux). L'effet de l'interaction montre un cercle candidat.



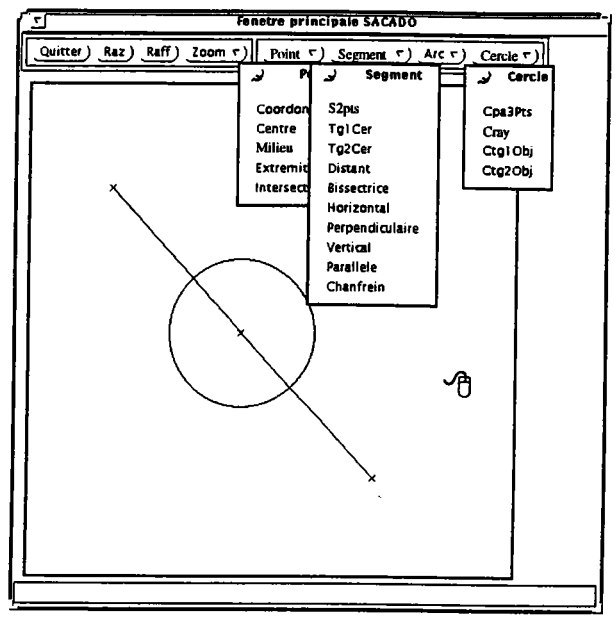
L'opérateur a sélectionné le menu terminal MILIEU qui est local. On remarque que l'effet continue car le menu est local. Il faut à présent construire le segment entre les deux points.



L'opérateur a sélectionné le menu terminal S2PTS qui est immédiat de par son père. On remarque que l'effet précédent s'est arrêté et est remplacé par un autre. Ici, le premier point du segment a déjà été désigné par l'opérateur.



La construction du segment est terminée, l'action globale du menu terminal MILIEU est réactivée. L'opérateur doit donc maintenant désigner le segment dont il désire prendre le milieu.



La construction du cercle est terminée, le milieu a été calculé et envoyé à l'action de construction du cercle qui s'est terminée à son tour. Le système attend une nouvelle sélection de menu ...

Dans la maquette, les compatibilités sont représentées par des couleurs différentes indiquant clairement le rôle de chaque menu en fonction du contexte.

SACADO a également servi de base à la construction d'un système basé sur les contraintes [GAR 95a]. Dans ce système, les contraintes sont ajoutées par une calculatrice grapho-numérique intégrée aux dialogues. Cette calculatrice, représentée par un domaine annexe (tous les boutons sont des menus de ce domaine annexe) est activée de la même façon que tout domaine annexe local. Il suffit de la déclarer locale aux interactions dont le résultat doit être obtenu à partir d'une expression. La figure ci-dessous montre la construction d'une contrainte : la longueur du segment 10 doit être égale à la distance entre le point 2 et le segment 8 augmentée de 10,2.

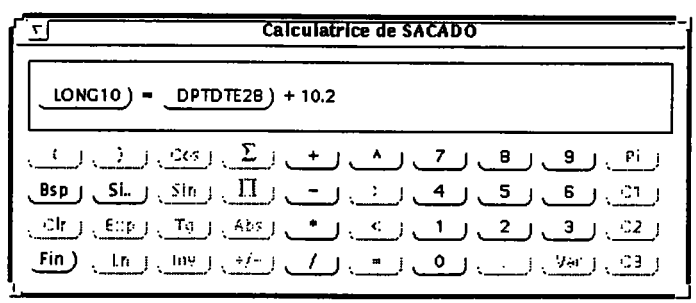


Figure II.17. *La calculatrice grapho-numérique.*

Ce développement a également permis de montrer que SACADO s'adapte aussi bien à la manipulation directe qu'à un dialogue de type verbe-objets. Ainsi, les contraintes sont détectées et manipulées dès la construction des objets sans faire appel à aucun menu, uniquement par manipulation de l'objet de l'intérêt.

6. CONCLUSION.

Ce chapitre propose une formalisation d'une nouvelle approche du dialogue homme-machine en C.F.A.O. Cette approche s'inscrit dans le cadre du projet SACADO développé au L.R.I.M. L'objectif était de fournir à la fois un modèle d'interaction et une méthodologie de développement pour le développement de l'architecture d'une application C.F.A.O.

Ce système, basé sur des menus, possède un modèle d'interaction dans lequel chaque commande est décrite comme un objet séparé appelé action globale que l'on peut assimiler en première approche à un sous-programme. Chaque action globale possède une description indépendante du dialogue que l'on peut décrire par un graphe d'actions interactives ou non. Le modèle utilisé autorise une action globale à être suspendue ou reprise grâce à l'utilisation d'un état courant.

Alors que les actions non interactives sont des appels aux primitives de l'application, les actions interactives ont pour but de gérer des interactions complexes avec l'opérateur. Dans la mesure où notre objectif était de laisser un maximum de liberté à l'opérateur, nous avons utilisé une primitive unique de dialogue appelée INTERACTION et des opérateurs de composition. Il est très facile d'implémenter des dialogues complexes sans aucune programmation, simplement en paramétrant les primitives de dialogue. De plus, la primitive unique garantit des dialogues homogènes entre différentes implémentations et facilite l'apprentissage pour un opérateur. Il a toujours à sa disposition les mêmes possibilités pour répondre quelle que soit l'interaction.

Des travaux se sont également intéressés à un composant de base appelé interacteur pour décrire les interactions de l'opérateur [FAC 92] [DIN 93] [HUB 89]. Cependant, ces interacteurs sont essentiellement destinés à la modélisation du composant *Présentation* car ils n'offrent que peu de contrôle sur l'application (voire aucun). D'ailleurs, les exemples fournis à partir de ces approches décrivent le fonctionnement d'éléments d'interface tels que des ascenseurs, des boîtes de dialogue, ... Il est à noter que de tels interacteurs sont une solution élégante pour représenter le composant *effets* des interactions. Cette description est certainement une perspective intéressante pour augmenter l'indépendance entre le dialogue et l'application.

Les actions globales sont ensuite combinées à l'aide de compatibilités pour définir l'ensemble de l'interface utilisateur plutôt que de la définir globalement. Les compatibilités définissent une organisation sémantique transversale des menus à la hiérarchie des menus. Cette définition des compatibilités permet un contrôle des fils d'activités multiples de l'opérateur et indique les menus autorisés en accord avec l'application. L'opérateur peut ainsi se situer et se repérer par rapport à sa tâche. Un contrôleur simple est défini pour gérer le flot de contrôle et garantit qu'il n'y a qu'une seule action globale active à un instant donné. Il suspend les action globales, pour passer le contrôle aux autres, dans le respect de la définition des compatibilités. Des mécanismes d'héritage et de transformation des compatibilités sont fournies pour éviter des répétitions inutiles dans la spécification, répétitions qui nuiraient à l'indépendance mutuelle des actions globales. Par le jeu des compatibilités, SACADO peut à la fois être considéré comme un système modal et un système non modal.

Lorsqu'aucune compatibilité n'est définie, une interaction d'une action globale sera toujours activée avec le même état de l'interface d'où l'appellation de système modal. Mais si des compatibilités sont définies, une interaction sera activée avec un état dépendant du raisonnement suivi par l'opérateur. Dans ce cas, le système devient non modal.

Cet apport des compatibilités se retrouve dans deux systèmes issus de la recherche : TIGER [KAS 82] et Chisl [WOO 88]. Dans le système TIGER, toute commande est implicitement considérée comme incompatible avec les autres sauf indication contraire. Le système Chisl utilise un arbre de menus. Il offre alors la possibilité de déclarer des menus globaux par rapport aux sous-arbres, l'action associée à ces nœuds globaux pouvant être sélectionnée à partir de n'importe quel niveau inférieur. De plus, une "retraite sélective" est mise en place par la possibilité d'annuler un menu sélectionné préalablement. Alors que ces deux systèmes ne supportent pas la notion de local, il est possible d'obtenir les mêmes possibilités avec SACADO par une définition judicieuse des compatibilités. Les commandes incompatibles de TIGER sont obtenues en les définissant différées entre elles, un menu global de Chisl peut être décrit par un menu immédiat et un menu différé dès sa sélection aurait le même comportement que la "retraite sélective". De plus, il est possible de définir ces compatibilités de manière plus fine en descendant au niveau des interactions.

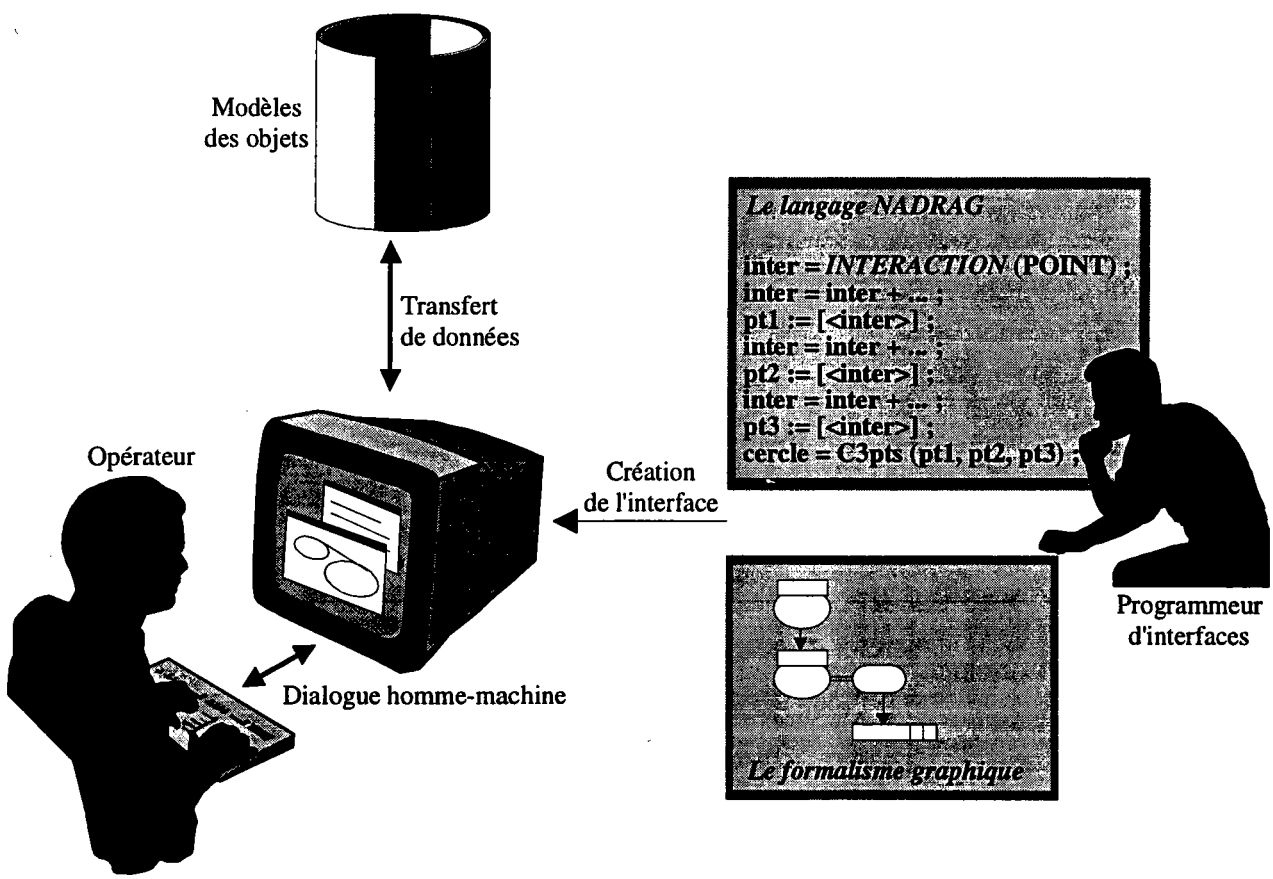
Le modèle d'interaction ainsi proposé s'inscrit comme une base vers une synthèse des réseaux de transition d'états, de l'approche multi-agent et de l'approche orientée objets. Une action globale constitue un réseau de transitions dont les nœuds sont les actions. Chaque action est un agent indépendant communiquant avec les autres grâce à un contrôleur qui maintient les fils d'activités. Ce contrôleur garantit la cohérence du dialogue et surtout l'héritage des compatibilités des interactions à travers l'évolution des menus.

La méthode de développement d'un système de C.F.A.O. à partir du noyau SACADO passe donc par deux phases principales. La première, qui est la partie statique, est constituée par la définition des menus et des actions globales du système. Les actions globales sont construites indépendamment les unes des autres et sont testées ainsi. La seconde passe par la définition des compatibilités, c'est la partie dynamique du développement. Le programmeur d'interfaces construit peu à peu les autorisations et les chemins que va pouvoir emprunter l'opérateur pour construire son objet graphique.

Dans les deux chapitres qui suivent, nous nous intéressons plus particulièrement à la construction des actions globales, que ce soit à l'aide d'un langage textuel ou d'un formalisme graphique de description d'actions ou de sa génération à partir d'informations provenant du modèle générique.

CHAPITRE 3.

LES MODÈLES DE DIALOGUE DE SACADO.



1. INTRODUCTION.

Jusqu'à présent, nous avons tenté d'apporter des solutions à l'utilisation d'un système de C.F.A.O. dans le respect des objectifs que nous avions fixés. Notre modèle d'interaction fournit un langage à l'opérateur pour communiquer avec le système (menus, compatibilités, ...). Le fait que les actions réalisées sont déterminées par les entrées de l'opérateur indique clairement que sa participation est très importante dès les premières étapes du développement [WAS 85]. Il peut fournir des informations capitales.

Cependant, malgré cette constatation, les logiciels satisfont rarement tous les besoins de l'opérateur. D'un côté certains systèmes se contentent de mettre en place des mécanismes de personnalisation (couleurs, ...) et d'extension simple (macro commandes, ...). D'un autre côté, l'opérateur est amené à programmer son application, ce qui est un véritable challenge intellectuel. Il ne faut toutefois pas perdre de vue la différence de tâche qui existe entre un opérateur et un programmeur. Un opérateur désire généralement résoudre un problème spécifique auquel il s'intéresse alors qu'un programmeur possède un domaine plus large. Il cherche des solutions générales et doit prendre en compte d'éventuelles évolutions du logiciel.

Trop souvent, l'opérateur est dominé par l'application qu'il utilise. Son espace de conception est limité par la vision du programmeur. Au lieu de diriger l'application, l'opérateur est souvent conduit et contraint par l'application. De notre point de vue, il est souhaitable de rapprocher les trois utilisateurs de SACADO pour leur offrir des outils similaires de développement et profiter au maximum des connaissances de chaque intervenant.

Nous pensons que l'utilisation de bibliothèques est un premier pas vers une approche simplifiée de la programmation. On peut trouver notamment :

- une bibliothèque d'interactions. Elle contient les données génériques d'une interaction et fait référence à la structure des menus pour gérer la prise en compte des compatibilités. Cette bibliothèque soulage le programmeur d'interfaces de descriptions fastidieuses et répétitives tout en harmonisant le dialogue du système. Toutes les interactions sont alors réalisées selon des moules identiques (elles "héritent" des interactions de la bibliothèque), ce qui place l'opérateur dans un contexte connu (peu de variations dans les possibilités offertes pour une même catégorie d'interactions);
- une bibliothèque d'actions non interactives. Elle recense les traitements qui ne font pas appel à des dialogues. Cette bibliothèque gère également les demandes de développement et les choix d'implantation. On rejoint l'idée qui consiste à laisser le soin à un non-informaticien de définir les actions globales en faisant appel à des spécialistes de l'informatique pour des cas bien particuliers;
- une bibliothèque d'actions globales. Elle contient le "découpage" en interactions et actions non interactives. Le programmeur d'interfaces dispose avec cette bibliothèque d'une véritable mine

d'informations, elle contient toutes les actions globales existantes dans l'application. Il peut s'en inspirer ou analyser leur réalisation pour en développer d'autres;

- une bibliothèque des menus. Elle permet essentiellement de représenter l'arborescence du dialogue par l'intermédiaire de l'arborescence des menus et d'associer les actions globales aux menus terminaux. Chaque action globale qui est associée aux menus terminaux fait référence à la bibliothèque des actions globales.

Dans ce chapitre, nous nous intéressons à ces deux dernières bibliothèques et, plus particulièrement à l'outil de génération des menus et au langage de définitions des actions globales (langage d'actions). Les outils que nous proposons forment un système générique qui définit l'architecture complète d'une application à travers un ensemble de spécifications.

2. LE GÉNÉRATEUR DE MENUS.

Notre premier travail a été de définir un outil graphique de description des menus destiné au programmeur d'interfaces. Cet outil doit permettre de modéliser interactivement les objets et les relations qu'il manipule, à savoir :

- des menus;
- des structures hiérarchiques;
- des liens de compatibilités entre menus.

Comme il est indiqué dans le chapitre précédent, les objets qui décrivent le monde du programmeur d'interfaces sont des domaines constitués de menus, menus auxquels sont associés des liens de compatibilités (immédiat, local, différé, inactif) et des liens hiérarchiques de type père-fils.

Pour rester fidèle à notre objectif de départ, nous considérons que cet outil n'est rien d'autre qu'une implantation SACADO dont l'opérateur est le programmeur d'interfaces. Il se présente comme toute autre application et bénéficie de tous les concepts que nous avons introduits. Il est souhaitable que tous les outils qui gravitent autour de SACADO se présentent comme une implantation SACADO. Leur intégration pourra se faire dans toutes les applications par une simple fusion des domaines de menus; les outils seront alors des options des applications ce qui permettra à tout moment d'en modifier la spécification. La seule différence réside dans le domaine d'application. C'est un exemple complexe (non C.F.A.O.) qui renforce la généralité de notre modèle de dialogue. L'implémentation du générateur de menus représente l'aboutissement des travaux sur les menus effectués en commun dans le cadre d'une thèse d'ingénieur C.N.A.M. [VOY 92].

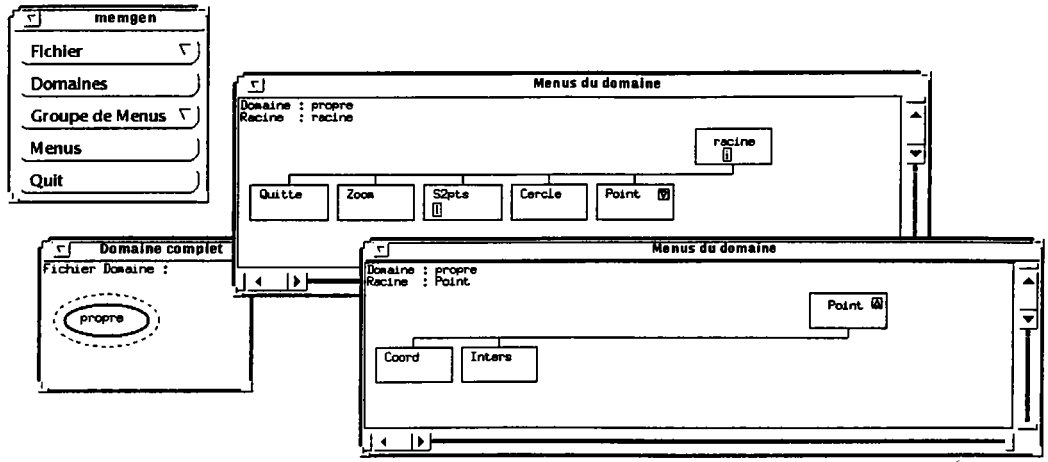


Figure III.1. *L'outil de description des menus.*

Dans la figure précédente, on voit apparaître la hiérarchie de menus que l'on peut rapprocher de l'exemple de la figure II.10. On y trouve un seul domaine, le domaine propre constitué de cinq menus principaux : QUITTER, ZOOM, S2PTS, CERCLE er POINT. Le menu POINT possède deux fils, les menus COORD et INTERS. Il est possible d'indiquer des compatibilités entre menus, par exemple :

- ZOOM immédiat de tous les menus du domaine propre. Le menu ZOOM est donc défini comme immédiat de la racine et devient accessible par héritage père-fils dans toutes les actions globales du domaine propre. Le "i" qui apparaît dans le menu racine permet d'accéder aux immédiats déjà définis, ZOOM en particulier;
- POINT local du menu S2PTS. Le menu POINT est donc défini comme local de S2PTS et, par suite de toutes les interactions de l'action globale associée. Le "l" qui apparaît dans le menu S2PTS permet d'accéder aux locaux définis, POINT en particulier.

Il est à noter que l'ajout de ces compatibilités se fait selon la représentation par graphes, indépendante du contexte, comme nous l'avons préconisée lors de la description du modèle d'interaction.

Bien entendu, ce générateur ne peut à lui seul définir le générateur de dialogue et d'architecture. Les menus sont parties prenantes dans le dialogue mais ils n'en forment qu'une partie. Notre propos étant de définir l'architecture de l'application à travers le dialogue, il nous faut donner au programmeur d'interfaces les moyens de décrire les actions globales de SACADO.

3. VERS UN LANGAGE D'ACTIONS.

Notre travail met l'accent sur la recherche d'un langage pour spécifier le dialogue d'une application de C.F.A.O. En particulier, nous nous intéressons à la complexité statique des logiciels et à leur complexité dynamique d'exécution. L'abstraction et la réutilisation sont deux facettes incontournables d'un tel langage pour qu'il soit de suffisamment haut niveau pour éviter à son utilisateur de traiter des détails qui ne le concernent pas.

Il s'agit de combler le gouffre qui existe entre le prototypage et la production finale (prototype et produit fini). Généralement, les outils de prototypage ne supportent pas ou mal la programmation à

grande échelle [MYE 92b]. C'est un problème difficile à résoudre lorsque le programme définitif doit s'exécuter rapidement, comme c'est souvent le cas en C.F.A.O.

Par la suite, nous proposons deux solutions que l'on peut considérer comme complémentaires :

- un langage textuel pouvant servir à la fois pour la construction du dialogue et de l'application. Ce langage possède des caractéristiques de haut niveau pour le prototypage rapide (interprétation, intructions simples et peu nombreuses, ...) mais aussi des fonctionnalités de bas niveau pour obtenir une application efficace (traducteur vers un langage compilable, instructions à vocation "professionnelle", ...);
- un langage graphique pour permettre à un utilisateur peu habitué à la programmation conventionnelle (l'opérateur et le programmeur d'interfaces dans de nombreux cas) de participer au développement. Il permet un développement rapide de l'architecture des actions globales.

Ces deux langages ont en commun le fait d'utiliser le dialogue comme trame de développement. Le but principal est de faciliter au maximum l'accès aux concepts de SACADO pour réduire les difficultés de programmation, notamment pour le côté dynamique de l'exécution, de telle sorte qu'ils soient accessibles par les trois utilisateurs : l'opérateur, le programmeur d'interfaces et le programmeur d'applications et de modèles.

4. UN LANGAGE TEXTUEL : NADRAG.

La syntaxe de ce langage s'inscrit dans la continuité des travaux réalisés sur la définition d'un langage pour la description de pièces paramétrées. Pour mémoire, en C.F.A.O., on est souvent amené à utiliser des pièces déjà définies antérieurement. Pour éviter d'avoir à redessiner entièrement la pièce, il est souhaitable de constituer une bibliothèque contenant les pièces à usage répétitif. Cela permet de rappeler à volonté les pièces mémorisées et donc de concevoir des objets plus rapidement. Le terme paramétré provient du fait que l'on peut obtenir, à partir d'une seule définition géométrique de la pièce, des pièces de dimensions différentes à partir de paramètres à préciser.

Dans de tels cas, mais aussi pour l'écriture de nouveaux traitements applicatifs, les logiciels de C.F.A.O. font le plus souvent appel à des langages de programmation classiques tels que LISP ou C++. L'utilisation de ces langages est rendue possible par la fourniture d'un logiciel de base adapté : primitives de création d'éléments géométriques, primitives d'affichages graphiques, fonctions de calculs, ... Toutefois, cette approche manque quelque peu de souplesse, étant donné qu'elle nécessite un programmeur spécialisé en ce langage, alors qu'un intermédiaire n'est pas, en général, souhaitable.

Notre but est de proposer au programmeur d'interfaces un langage simple à utiliser mais qui offre la possibilité de réaliser n'importe quel type de traitements même complexe. La spécification des actions globales se fait de manière textuelle et sous une forme lisible. Nous avons choisi cette représentation car la représentation textuelle reste encore la façon la plus efficace pour écrire des traitements complexes [MYE 92b]. En effet, bien qu'une représentation graphique puisse être adaptée pour

représenter des interactions graphiques, de nombreux traitements restent du domaine textuel (problèmes de représentation des calculs, des itérations, ...). L'introduction d'un formalisme graphique nous permettra de choisir la représentation la mieux adaptée à tout moment.

L'exécution de NADRAG passe par l'utilisation d'un interpréteur. Ce choix se justifie par un besoin rapide de résultats. Le programmeur d'interfaces doit pouvoir exécuter une action globale, et donc visualiser son résultat dès son écriture. Cela lui permet de vérifier immédiatement que sa description est correcte et de corriger les erreurs le cas échéant. Lors de la mise au point, une phase de compilation n'apporterait que des inconvénients malgré les progrès effectués sur la compilation incrémentale et les liens dynamiques. Nous n'excluons pas qu'à terme ces actions globales interprétables soient transposées en un langage plus classique. Elles seront alors compilées et incluses automatiquement dans le logiciel. Une première étude dans ce sens a été réalisée en commun dans le cadre d'une thèse d'ingénieur C.N.A.M. [HIG 94].

Nous présentons dans les paragraphes suivants les étapes de conception et d'implémentation de NADRAG [GAR 92] [MAR 92b]. Cette étude principalement axée sur les instructions du dialogue se décompose en deux phases principales :

- définition syntaxique du langage pour la construction d'une action globale tout en respectant le modèle d'interaction de SACADO;
- développement d'un interpréteur pour mettre en œuvre le langage. Il a pour rôle d'exécuter une action globale donnée et de visualiser la construction obtenue.

4.1. Les structures de données.

La notion de variable est indispensable pour pouvoir s'appuyer sur des définitions réalisées antérieurement. Un débat existe dans la littérature sur l'importance ou non d'un typage fort [MYE 92b] : les uns affirment que l'on ne peut pas s'en passer pour des applications complexes car il facilite la détection et la prévention des erreurs alors que les autres argumentent sur la contrainte et sur la perte de temps induites par la déclaration des types. Dans la mesure où notre langage est destiné à des programmeurs d'interfaces et à des opérateurs, nous avons choisi un compromis entre ces deux points de vue : le type est inféré plutôt qu'indiqué. Une variable est créée immédiatement après sa première utilisation et typée par l'objet affecté. C'est possible en particulier grâce aux constantes scalaires (100.0, 23, ...) et à des constructeurs d'objets, comme dans le cas des instructions du dialogue.

Nous devons préciser qu'une variable change de type lorsqu'un objet d'un autre type est affecté à une variable existante. La syntaxe de l'affectation est la suivante :

Nom_Variable = *Expression* ;

Par analogie aux variables, les tableaux n'ont pas à être déclarés. Ils le sont dès la première utilisation et leur taille est également inférée. Par exemple, une instruction 't [1] = 10.0 ;' déclare implicitement un

tableau à une dimension. Notons que chaque élément d'un tableau est une variable à part entière, ce qui autorise plusieurs types différents dans un même tableau.

4.2. Les instructions graphiques.

Généralement, les opérations réalisées sur les objets sont :

- leur définition (création);
- leur visualisation (affichage).

Pour un objet donné, ces deux opérations se suivent lors de sa construction. Il a donc été décidé de les lier en les confondant en une seule instruction, l'affectation. Le fait de définir l'objet provoque son affichage [SLO 89].

Les instructions graphiques comportent donc un membre gauche et un membre droit séparé par le signe '='. Le membre droit est évalué et sa valeur est rangée dans l'objet figurant dans le membre gauche. Les objets que l'on peut rencontrer sont les objets manipulés par SACADO. Par exemple, on peut avoir :

objet	construction
POINT	abscisse (SCALAIRE), ordonnée (SCALAIRE)
SEGMENT	deux extrémités, origine et fin (POINT)
CERCLE	centre (POINT), rayon (SCALAIRE)

Les paramètres de construction de la table précédente indique les valeurs que l'on doit spécifier dans le membre droit de l'affectation pour créer l'objet de façon cohérente. Par exemple :

```
point   = milieu (segment) ;              affectation directe objet / objet
point   = POINT (100.0 , 200.0) ;         création d'un point par abscisse et ordonnée
segment = SEGMENT (point1 , point2) ;     création d'un segment s'appuyant sur deux points
segment = tangent_2_cercles (cercle1 , cercle2) ; création d'un segment tangent à deux cercles.
                La fonction tangent_2_cercles peut contenir des dialogues pour choisir la
                solution. Elle est donc vraisemblablement écrite en NADRAG.
```

Les instructions graphiques ne sont pas réservées aux objets de base, on peut également rencontrer des objets complexes, tels que des cotations ou des hachurages :

objet	construction
COTATION	deux objets, obj_1 et obj_2 (POINT ou SEGMENT)

Ainsi, l'instruction "cote = COTATION (point, segment) ;" entraîne la création d'une cote entre le point 'point' et le segment 'segment'.

4.3. Les instructions du dialogue.

La manipulation du dialogue repose sur trois instructions principales :

- l'affectation (description des interactions à mettre en œuvre);
- l'activation (l'interaction est prise en charge par le système et l'opérateur doit répondre en accord avec les spécifications indiquées);
- l'accès au résultat (l'interaction a été validée et on désire connaître la réponse de l'opérateur).

Nous définissons de manière générale ces trois instructions avant d'en détailler le principe pour chacune des interactions (INTERACTION, ET, OU et ENSEMBLE).

Comme nous l'avons indiqué dans le chapitre précédent, une interaction est formée d'un ensemble de composants qui doivent être paramétrés pour obtenir l'interaction désirée.

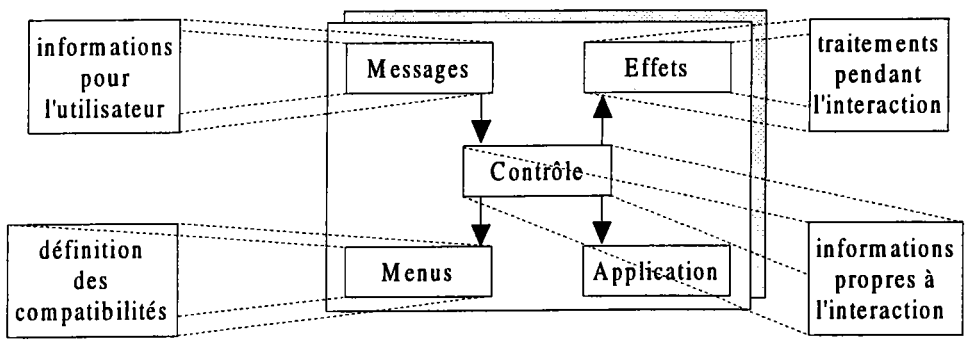


Figure III.2. *Les composants d'une interaction.*

Nous avons donc choisi de construire une interaction comme une collection d'objets que nous appelons des objets du dialogue. Ces objets représentent toutes les possibilités dont nous disposons pour paramétrer une interaction :

- les messages;
- les effets;
- les restrictions sur les objets (*Contrôle*);
- les restrictions sur les menus.

Pour chacun de ces objets, nous décrivons un ou plusieurs constructeurs.

Les messages sont des informations visuelles fournies à l'utilisateur pour l'aider à répondre à l'interaction. Dans notre cas, nous considérons uniquement des aides textuelles à l'aide du constructeur suivant :

***MESSAGE* (Texte)**

Exemple : *MESSAGE* ("Désigner le point origine du segment")

Les effets définissent des traitements à effectuer lorsque l'interaction est active. Ces traitements ont pour but d'aider l'opérateur à répondre à l'interaction. Contrairement aux messages qui définissent une aide statique, les effets sont utilisés pour fournir une aide contextuelle alors que l'interaction se précise. Le constructeur s'appuie sur des instructions NADRAG comme le ferait un sous-programme :

***EFFET* (instructions)**

Exemple : *EFFET* (segment = SEGMENT (origine, position courante) ;)

Les restrictions sur les objets sont des contraintes que le résultat d'une interaction doit vérifier. Généralement, ces conditions expriment le domaine de validité du ou des objets demandés à l'opérateur en fonction du contexte d'exécution. Ce sont des expressions à valeur booléenne :

***CONTRAINTE* (Expression booléenne)**

Exemple : *CONTRAINTE* (rayon > 0)

Les restrictions sur les menus permettent de restreindre les différentes sélections possibles de menus. Elles s'appliquent aux menus de l'application concernée ou aux domaines (menu et domaine sont entendus au sens large, l'indication d'un menu vaut pour tous ses fils et celle d'un domaine pour tous ses menus). Nous fournissons donc des constructeurs pour les objets de l'interface (menu et domaine) et pour les compatibilités :

DOMAINE	**(Nom_Domaine)**	construction du domaine dont le nom est Nom_Domaine
MENU	**(Nom_Menu)**	construction du menu dont le nom est Nom_Menu
IMMEDIAT	**(Domaine ou Menu)**	construction d'une compatibilité immédiate
LOCAL	**(Domaine ou Menu)**	construction d'une compatibilité locale
DIFFERE	**(Domaine ou Menu)**	construction d'une compatibilité différée
INACTIF	**(Domaine ou Menu)**	construction d'une compatibilité inactive

Pour décrire une interaction, il suffit de combiner tous ces objets du dialogue. Pour cela, nous définissons deux opérateurs de construction, + et -. Il est possible d'ajouter (+) ou de retirer (-) des objets du dialogue à une interaction donnée. Les interactions se trouvent alors précisées grâce à l'affectation. Soit Inter une interaction, on peut trouver par exemple :

Inter = Inter + *IMMEDIAT* (*DOMAINE* ("Propre")) ;
le domaine propre se retrouve immédiat de *Inter*

Inter = Inter - *LOCAL* (*MENU* ("Point")) ;
le menu Point ne doit pas être local de *Inter* même s'il l'a été précédemment

Bien entendu, les opérateurs de construction s'appliquent à tous les types d'interactions et le résultat est une interaction de même type.

Cette définition des interactions par un ensemble d'objets du dialogue présente des avantages non négligeables. Il est très facile d'ajouter un nouvel objet (et donc de nouvelles possibilités) sans remettre en cause les actions globales déjà décrites. De plus, il est possible de réutiliser une interaction déjà définie pour en construire une autre. On peut faire partager des objets communs à plusieurs interactions (on s'intéresse aux objets du dialogue qui varient d'une interaction à l'autre) pour faciliter les modifications des actions globales.

L'interaction étant construite, il faut l'activer. La communication avec l'utilisateur se fait grâce à une instruction simple d'entrée. Soit *Inter* une interaction définie au préalable, l'instruction **< *Inter* >** entraîne l'activation de l'interaction *Inter* jusqu'à validation ou annulation.

Après validation, l'accès au résultat d'une interaction ne peut se faire directement par l'identificateur de l'interaction, le système ne peut faire la différence entre l'interaction elle-même et son résultat. Pour faciliter la tâche d'apprentissage, nous avons donc choisi une instruction dont la syntaxe est proche de l'instruction d'activation. Soit *Inter* une interaction définie au préalable, l'instruction **[*Inter*]** représente le résultat de l'interaction *Inter*. Notons qu'un effort de simplification est introduit à ce niveau; l'instruction *<Inter>* est implicite par défaut. Lorsque l'on rencontre un identificateur d'interaction non activée, l'interaction correspondante est automatiquement activée, le dialogue est alors initié par le système.

Nous illustrons ces différents concepts pour chaque type d'interaction à travers l'étude détaillée de plusieurs exemples.

4.3.1. La primitive INTERACTION.

Nous rappelons que cette primitive "élémentaire" a pour but de récupérer un objet à partir d'un certain nombre de contraintes. L'instruction de création n'a pas pour but de créer entièrement l'interaction mais de construire une interaction minimale que l'on enrichira par la suite à l'aide des opérateurs de construction et des objets du dialogue. Nous utilisons donc une instruction qui définit une interaction à partir du type de l'objet désiré :

Identificateur_Interaction = *INTERACTION* (Type_de_l'objet_désiré) ;

Par exemple, considérons la construction d'un cercle à partir d'un centre et d'un rayon positif significatif :

centre = *INTERACTION* (POINT) ;
centre = centre + *LOCAL* (*MENU* ("Point")) ;
rayon = *INTERACTION* (*SCALAIRE*) ;
rayon = rayon + *CONTRAINTE* ([rayon] > ε) ;
<centre> ;
<rayon> ;
cercle = CERCLE ([centre], [rayon]) ;

Il faut remarquer que pour avoir accès ultérieurement au résultat d'une interaction, il est nécessaire qu'elle ait été construite au préalable à l'aide d'une variable. En effet, l'instruction <*INTERACTION* (POINT)> est correcte mais on ne peut pas accéder à son résultat dans une instruction ultérieure, étant donné que l'on ne peut y faire référence. Il est souhaitable d'utiliser des variables, cela facilite la compréhension et ne pénalise pas la programmation dans la mesure où il est inutile de les déclarer.

La construction d'un cercle passant par trois points est l'exemple type d'interactions peu différentes les unes des autres. Grâce à notre approche modulaire, nous allons pouvoir définir un "patron" d'interaction, qui sera précisé au fur et à mesure de la construction de l'action globale :

patron = *INTERACTION* (POINT) ;
patron = patron + ... ;
point [1] = [<patron>] ;
patron = patron + *CONTRAINTE* ([patron] ≠ point [1]) ;
point [2] = [<patron>] ;
patron = patron + *CONTRAINTE* ([patron] ≠ point [2])
+ *CONTRAINTE* ([patron] ∉ Droite (point [1], point [2])) ;
point [3] = [<patron>] ;
cercle = CERCLE (point [1], point [2], point [3]) ;

Cette action montre pourquoi il est parfois nécessaire d'activer explicitement une interaction. L'interaction *patron* a déjà été validée pour le premier point lorsqu'elle est réactivée pour le deuxième point. Il est donc nécessaire d'indiquer au système et ce, explicitement, que l'on désire un autre point et non consulter sa valeur actuelle.

La construction des instructions de dialogue nécessite l'apprentissage de peu d'instructions et il est facile de créer des actions simples. L'introduction des opérateurs de composition va nous donner la possibilité de décrire des dialogues relativement complexes toujours de manière aussi simple.

4.3.2. Les opérateurs de composition.

Les opérateurs de composition sont utilisés pour combiner les interactions élémentaires. On retrouve donc une instruction de construction pour chacun des opérateurs :

Opérateur = *ET* (Inter$_1$, .. , Inter$_n$) avec $n \geq 0$
Opérateur = *OU* (Inter$_1$, .. , Inter$_n$) avec $n \geq 0$
Opérateur = *ENSEMBLE* (Inter)

L'ajout ou le retrait des interactions peut également se faire par l'intermédiaire des opérateurs de construction. On ajoute une interaction à un opérateur de composition grâce à l'opérateur de construction '+' et on en retire une avec l'opérateur de construction '-'.

Dans notre exemple du cercle construit à partir d'un centre et d'un rayon, nous avions fixé l'ordre (centre puis rayon). Pourtant, cet ordre n'est pas obligatoire. Grâce à l'opérateur ET, nous pouvons construire le cercle dans n'importe quel ordre :

```
centre = INTERACTION (POINT) ;
centre = centre + LOCAL (MENU ("Point")) ;
rayon = INTERACTION (SCALAIRE) ;
rayon = rayon + CONTRAINTE ([rayon] > ε) ;
inter = ET (centre, rayon) ;
inter = inter     + EFFET ( réticule )
                  + IMMEDIAT (MENU ("Zoom")) ;
<inter> ;
```

Pour avoir accès aux différents résultats des interactions, il faut pouvoir accéder aux interactions qui composent le ET. Il convient de mettre en place une instruction pour extraire les constituants : **NomOpérateur.NomInteraction**. Cette instruction illustre bien la hiérarchie d'interactions que nous avons étudiée au chapitre 2. L'action de création du cercle prend alors la forme :

```
cercle = CERCLE ([inter.centre], [inter.rayon]) ;
```

L'opérateur OU est plus simple dans la mesure où il ne possède qu'un seul résultat, celui de l'interaction validée. Pour y accéder, il suffit d'utiliser l'instruction **[Nom_Opérateur]** ce qui a pour conséquence de donner le résultat de l'interaction validée.

Il faut noter qu'à la différence des autres interactions, la valeur d'un opérateur ENSEMBLE est un tableau contenant les valeurs des interactions. Afin d'illustrer ce point, nous prenons comme exemple la construction d'un ensemble de points :

depart = *INTERACTION* (POINT) + ... ;
<depart> ;
point = *INTERACTION* (POINT) ;
polygone = *ENSEMBLE* (point) ;
point = point + *CONTRAINTE* ([point] ∉ [polygone]) ;
polygone = polygone + *CONTRAINTE* ([depart] = dernier ([polygone])) ;
<polygone> ;

La fonction dernier () est une fonction donnant accès au dernier point donné par l'opérateur. Lorsque celui-ci est égal au point de départ, l'interaction polygone est validée et [polygone] contient alors les points { pt_1, .. , pt_n, depart } dans l'ordre des réponses. Pour accéder au point numéro i, il suffit d'utiliser l'instruction [polygone] [i].

L'introduction des opérateurs de composition ne modifie que très peu les notations. Cela ne change en rien la complexité des actions globales qui suivent le même schéma syntaxique.

4.4. Les structures de contrôle.

Ces instructions permettent de modifier le flot de contrôle et donc l'ordre des instructions. Les structures de contrôle s'ajoutent au contrôle déjà inclus dans les interactions par les contraintes et les ensembles. Nous distinguons deux types de structures de contrôle :

- les instructions de décision (pour choisir entre plusieurs instructions);
- les instructions de répétition (pour répéter un ensemble donné d'instructions).

La première instruction de décision utilisée respecte le schéma suivant (les parties entre accolades sont optionnelles) :

SI Expression booléenne
ALORS instructions
{SINONinstructions}
FINSI ;

Lors de l'exécution, si l'expression booléenne est vraie, les instructions de la clause ALORS sont exécutées, dans le cas contraire ce sont les instructions de la clause SINON.

La seconde instruction de décision est plus complexe et est une extension de la première (les parties entre accolades sont optionnelles) :

CHOIX Expression
CAS $Expr_1$: instructions
... ...
CAS $Expr_n$: instructions
{SINON: instructions}
FINCHOIX ;

Le mot réservé CHOIX est suivi d'une expression dont on désire tester le résultat. Lors de l'exécution, chaque cas est analysé dans l'ordre de sa définition. Si il existe une expression $Expr_k$ dont le résultat est égal à l'expression testée alors les instructions de ce cas sont exécutées. Dans le cas contraire, ce sont les instructions de la clause SINON qui sont exécutées.

La seule instruction de répétition que nous avons utilisée est l'instruction TANTQUE. Les deux schémas de construction sont :

TANTQUE Expression booléenne
instructions
FINTANTQUE ;

FAIRE
instructions
TANTQUE Expression booléenne ;

Dans le premier cas, si l'expression booléenne est vraie, les instructions sont exécutées puis on recommence lorsque l'instruction FINTANTQUE est rencontrée. Par contre, si l'expression est fausse, on exécute l'instruction suivant FINTANTQUE.

Dans le second cas, les instructions sont exécutées. Lorsque l'on rencontre TANTQUE, l'expression booléenne est évaluée et si elle est vraie alors on recommence. Dans le cas contraire, l'instruction qui suit le TANTQUE est exécutée. Comme exemple, nous donnons la construction d'un cercle à partir de trois points :

```
patron = INTERACTION (POINT) + ... ;
i = 1 ;
FAIRE
   point [i] = [<patron>] ;
   CHOIX i
   CAS  1 :  patron= patron  + CONTRAINTE ([patron] ≠ point [1]);
   CAS  2 :  patron= patron  + CONTRAINTE ([patron] ≠ point [2])
                             + CONTRAINTE ([patron] ∉ Droite (point [1], point [2]));
   FINCHOIX
   i = i + 1 ;
TANTQUE (i < 4) ;
cercle = CERCLE (point [1], point [2], point [3]) ;
```

Les structures de contrôle permettent au programmeur d'interfaces de construire différemment les interactions en fonction du contexte. Par exemple, les compatibilités peuvent varier en fonction de l'opérateur ou du contexte d'utilisation.

4.5. Les actions globales.

Un programme NADRAG représente une action globale d'un système. Son schéma de spécification est le suivant (les instructions entre accolades sont optionnelles) :

```
ACTION_GLOBALE Nom_Action ;

{blocs internes}

bloc principal
```

Le nom de l'action globale est le premier élément indispensable de la description. Il ne faut pas perdre de vue que dans la majorité des cas, un programme est associé à un menu terminal et comporte donc un objet comme résultat (ne serait ce que pour les liens locaux). Ce résultat est obtenu par une ou plusieurs instructions d'affectation. Il suffit d'utiliser la variable qui possède le nom de l'action globale et de lui affecter le résultat désiré :

Nom_Action = résultat ;

L'ossature d'un programme est clairement définie par un ensemble de blocs dans lesquels se trouvent les instructions. On trouve deux sortes de blocs :

- un bloc principal;
- des blocs plus internes, les procédures et les fonctions.

Le bloc principal représente le programme principal. L'exécution de l'action globale débute par ce bloc et il est défini par la structure suivante :

```
DEBUT
  instructions
FIN ;
```

Les blocs plus internes sont les sous-programmes de NADRAG. Ils sont de deux sortes : pré-définis et déclarés par le programmeur. Les exemples de blocs predéfinis sont nombreux, on peut citer les instructions graphiques. Les blocs utilisateurs doivent être déclarés avant le bloc principal mais peuvent être utilisés avant leur déclaration.

PROCEDURE (ou FONCTION) Nom (Param$_1$, .. , Param$_n$) ;
instructions
FINPROCEDURE (ou FINFONCTION) ;

La seule différence entre une procédure et une fonction réside dans le résultat. Pour indiquer ce résultat, nous procédons comme pour l'action globale, il suffit d'affecter le nom de la fonction avec le résultat souhaité. L'appel de tels blocs se fait simplement en indiquant son nom et ses paramètres.

4.6. Implantation et conclusion.

Une première version du langage NADRAG a été implantée et testée sur des stations SUN sous le système SUN OS 4.1.1. unix BSD4.3. Pour garantir une bonne portabilité et un développement plus rapide, il a été écrit à l'aide d'une grammaire et du langage C++ en respectant la norme ANSI. Les différents développements sont décrits dans l'annexe B.

NADRAG est un langage très simple comparé à d'autres approches destinées plus particulièrement aux programmeurs professionnels. C'est un langage impératif dans lequel la description des dialogues s'apparente plus à une description déclarative. Il est constitué d'un nombre restreint d'instructions de haut niveau et pourtant, sa puissance de description est grande dans la mesure où il satisfait notre modèle d'interaction.

Toute cette étude nous a amenés à considérer l'interaction comme un "objet" construit à partir d'autres objets et nous pensons que l'intégration de ces concepts dans un langage orienté objet plus classique est une suite intéressante. De cette façon, le dialogue deviendrait réellement le centre du développement. Un seul langage servirait pour le développement du dialogue et de l'application facilitant de cette façon l'intégration des différents composants. L'étude du transcodeur NADRAG vers C++ est considérée comme une première tentative pour évaluer la faisabilité de cette intégration.

Dans cette voie, un travail important reste à accomplir pour aboutir à un langage complet et exhaustif mais il nous semble que l'orientation choisie qui tend à se rapprocher de l'opérateur et de sa profession va dans le bon sens.

5. LA PROGRAMMATION VISUELLE.

Sous le terme de programmation visuelle, on retrouve tous les systèmes qui laissent l'opérateur spécifier un programme en utilisant une notation à deux dimensions. Par exemple, les langages conventionnels (textuels tels que NADRAG) ne sont pas considérés comme des langages à deux dimensions car le compilateur ou l'interpréteur traite les instructions comme un flot d'informations à une dimension.

Ainsi, les systèmes à programmation visuelle qui s'appuient sur des langages graphiques veulent répondre aux principales difficultés inhérentes aux langages conventionnels et notamment :

- la difficulté syntaxique qui provient de l'utilisation d'interfaces textuelles qui ne prévient pas les erreurs simples;
- la difficulté conceptuelle (par exemple le fait d'utiliser des variables fait apparaître une abstraction parfois difficile à comprendre);
- la difficulté de compréhension et d'appréhension des structures de contrôle qui indiquent une exécution non linéaire des actions.

En utilisant des langages graphiques, la productivité des programmeurs est augmentée par rapport aux techniques classiques. Par la manipulation d'objets graphiques mis en relation dans un espace à deux dimensions, on obtient une représentation plus proche du modèle conceptuel des programmeurs.

Dans le cadre de SACADO, l'outil graphique mis à la disposition de l'opérateur et du programmeur d'interfaces apparaît comme un environnement visuel interactif destiné à organiser un certain nombre d'interactions. En "dessinant" directement à l'écran, le programmeur d'interfaces décrit les interactions nécessaires à ses besoins et l'opérateur peut spécialiser des interactions déjà mises en place. De cette manière, le développement du système se transforme en une boucle fermée ou l'opérateur devient le programmeur à part entière de son environnement.

Le développement de ces langages graphiques est un paradigme de programmation relativement récent. Cependant, il faut bien avouer que les systèmes existants n'ont pas complètement convaincu [GIR 93; MYE 90b]. Notre challenge est de montrer que cette approche peut aider à résoudre des problèmes dans notre domaine et qu'elle est très bien adaptée aux concepts de SACADO. Dans ce but, nous présentons un formalisme pour notre modèle d'interaction et nous montrons comment son utilisation simplifie de manière significative le développement ou l'évolution d'une implantation SACADO.

5.1. Un langage graphique.

Les langages graphiques permettent la mise en place des liens logiques entre différents éléments sans devoir passer par l'écriture. Pour être facile à comprendre, ils doivent posséder une qualité essentielle : la simplicité.

L'élaboration d'un tel langage passe par une série d'étapes. Nous en distinguons quatre dans notre travail :

- s'interroger sur le public cible (voir l'interroger pour cibler ses besoins);
- analyser au préalable l'information à manipuler afin d'en exprimer une représentation;
- traiter l'information pour la convertir en éléments graphiques;
- s'enquérir de l'utilité d'une représentation graphique.

Nous abordons les deux premières étapes dans la suite de ce paragraphe. Nous présentons ensuite l'ensemble des éléments graphiques qui intervient dans ce formalisme avant d'en illustrer l'intérêt à travers divers exemples basés sur son utilisation.

Dans notre cas, l'opérateur peut être considéré comme un utilisateur professionnel. Il a une très bonne connaissance du domaine d'application mais possède a priori très peu de notions informatiques. Bien sûr, pour utiliser le formalisme, des notions lui sont indispensables et notamment sur notre modèle d'interaction. Nous donnons une liste non exhaustive des notions de base à connaître :

- les menus;
- les compatibilités;
- les interactions;
- ...

L'opérateur peut avoir acquis ces notions relativement simples au cours de l'utilisation de son outil de C.F.A.O.

Dès lors que le public visé est déterminé, il faut avoir une idée précise de l'information à transmettre pour concevoir notre formalisme. Le choix et l'utilité d'un type de formalisme dépendent du message que l'on désire transmettre. Le but de ce formalisme est de décrire les éléments graphiques de base du futur programme graphique. Pour notre formalisme, nous avons décidé d'adapter à notre problème les réseaux de transitions d'états. Cette représentation est déjà très utilisée dans d'autres travaux; elle fournit un outil rapide pour créer un dialogue et autorise une bonne représentation graphique pour un modèle de dialogue [BOR 92; SHE 92]. La notion d'état des réseaux convient très bien à la représentation de notre système, dont l'interaction avec l'opérateur fait largement usage de modes [COU 90], même s'ils sont dynamiques dans SACADO. À chaque mode correspond un contexte de travail (un contexte de menus dans notre cas). De plus, notre modèle d'interaction apporte une réponse aux deux principales critiques émises sur ces réseaux :

- les états doivent comporter des arcs explicites pour toutes les commandes universelles telles que l'aide, les corrections d'erreurs, ... Ces informations sont internes aux interactions par l'utilisation des compatibilités entres actions globales (par exemple, le menu aide est immédiat à tous les domaines ...);
- les réseaux deviennent vite très importants et difficilement compréhensibles. Le système SACADO est décomposé en actions globales, un diagramme décrit donc un traitement relativement simple et seuls les menus associés aux compatibilités décrivent le système complet.

Des utilisateurs non-programmeurs ou peu expérimentés, dont fait partie le public visé par notre outil, peuvent facilement participer à la définition du dialogue, car les réseaux sont faciles à comprendre et à utiliser.

Avec un tel formalisme, le passage d'un état à l'autre s'effectue le long des arcs étiquetés. Le système se trouvant dans un état donné et l'opérateur effectuant une action, l'arc suivi est celui dont les conditions d'activation sont satisfaites. Une action globale étant un graphe d'actions, nous décrivons dans les paragraphes suivants les nœuds et les arcs de notre formalisme. Les nœuds sont nos entités et les arcs sont les relations que l'on peut leur appliquer.

5.2. Les entités.

Comme nous venons de le voir, elles représentent les différentes actions que l'on peut rencontrer dans une action globale. Pour mémoire, on peut trouver :

- des interactions;
- des opérateurs de composition;
- des actions non interactives.

Toutes ces entités ont besoin d'un certain nombre de données, qui doivent être renseignées par le programmeur d'interfaces. Plusieurs possibilités lui sont offertes pour l'affectation de ces informations :

- de manière textuelle;
- par manipulation directe.

Dès que cela sera possible, nous insisterons sur la deuxième possibilité, qui offre une plus grande souplesse de fonctionnement pour des utilisateurs peu ou pas expérimentés.

Parmi toutes les entités introduites pour décrire une action globale, nous distinguons trois sortes d'entités particulières :

- une entité de départ. C'est le point d'entrée de l'exécution vis-à-vis de l'extérieur et elle est indiquée en traits forts dans notre formalisme;
- les entités d'arrivée. Ce sont les entités qui indiquent la fin de l'action globale. Elles sont implicites : il suffit qu'il n'y ait plus de possibilité de continuer l'exécution de l'action globale pour s'arrêter;
- les entités résultats. Elles donnent le résultat de l'action globale. Une action globale ne peut donner qu'un seul résultat, mais il peut y avoir plusieurs entités résultat, étant donnés les différents chemins que peut prendre un opérateur. Pour une exécution donnée, une seule entité fournira le résultat. Pour les distinguer, nous avons choisi de leur "accrocher" une enveloppe : ✉ (métaphore pour simplifier la compréhension du formalisme).

Par la suite, nous décrivons précisément les entités que l'on peut rencontrer dans notre formalisme.

5.2.1. Les interactions.

Chaque type d'interaction est représenté par un graphique différent, pour bien mettre en valeur les dialogues mis en place. Comme dans le cas de NADRAG, une interaction est décomposée en un ensemble de composants qui représente les informations spécifiques à lui associer. Les informations communes à toutes les interactions sont :

- les effets;
- les contraintes;
- les compatibilités;

- les messages.

Les effets et les contraintes sont des composants destinés à mettre en relation les entités du formalisme alors que les messages et les compatibilités sont véritablement des informations propres à l'interaction. Si les messages sont relativement aisés à indiquer, les compatibilités nécessitent un travail non négligeable. Dans NADRAG, l'utilisation des noms de menus ou de domaines entraîne une certaine lourdeur et surtout une dépendance nominative. Pour l'outil graphique, il est possible d'intégrer totalement la manipulation directe des menus dans la construction des compatibilités.

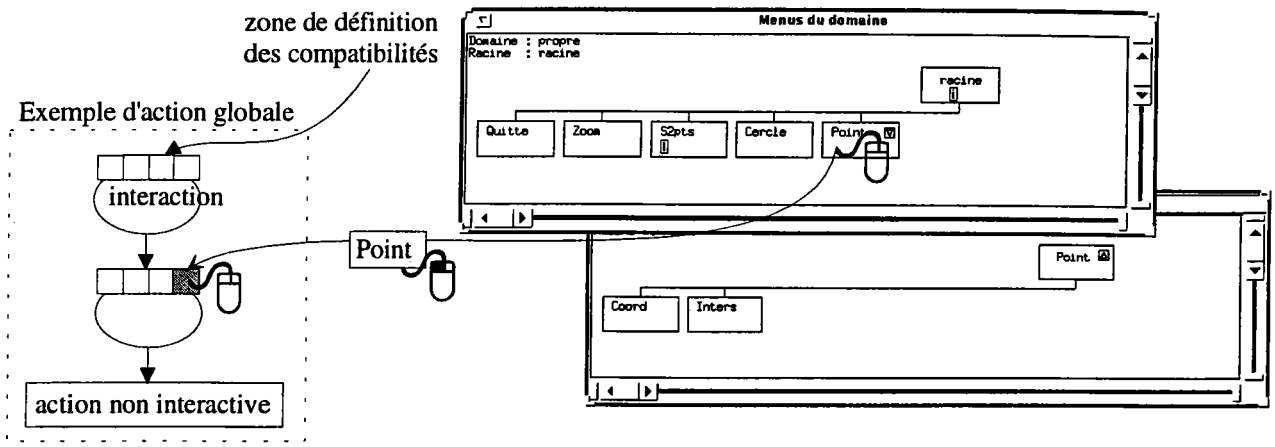


Figure III.3. *Définition d'une compatibilité après intégration du générateur de menus et de l'outil graphique.*

Ainsi, en sélectionnant directement les menus ou les domaines, le programmeur d'interfaces crée des liens de compatibilités sans se soucier des noms. Le fait que le générateur de menus et l'outil graphique soient tous les deux des implantations SACADO facilite bien entendu l'intégration : il est simplement nécessaire de fusionner les domaines de menus pour obtenir un système commun et de mettre en place les protocoles d'échange. Ces protocoles sont très importants. S'ils sont choisis convenablement, le changement du nom d'un menu ne doit pas dérouter les compatibilités (système multi-lingue), pas plus que le déplacement d'un menu dans l'arborescence.

5.2.1.1. La primitive INTERACTION.

C'est la primitive unique de dialogue entre l'opérateur et le système. Elle permet la désignation d'un objet et est définie par le type des objets accepté au cours du dialogue. Là encore, l'intégration de l'outil graphique permet de manipuler les types directement à l'écran en fonction des objets graphiques manipulables par l'application considérée.

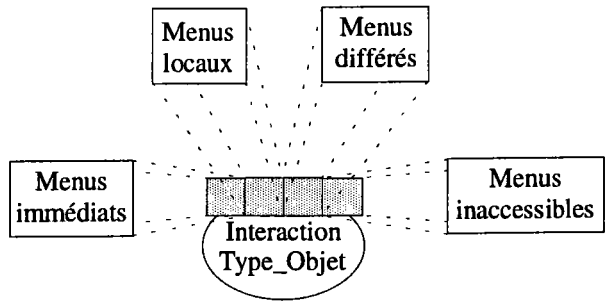


Figure III.4. *Une interaction de base.*

Les compatibilités n'apparaissent pas directement à l'écran pour ne pas surcharger la notation. Seules des indications apparaissent par lesquelles le programmeur d'interfaces peut consulter ou modifier les liens de compatibilités. C'est un choix qui se justifie car la construction d'une action globale et des compatibilités ne se situent pas au même niveau. Une action globale définit une action de l'application alors que les compatibilités spécifient les liens qui peuvent exister entre toutes les actions globales. Nous considérons donc qu'il existe un outil de plus haut niveau (vision globale de l'application) pour aider à mettre en place ces compatibilités.

5.2.1.2. Les opérateurs de composition.

Ces entités sont utilisées pour regrouper des interactions et former des dialogues de plus haut niveau.

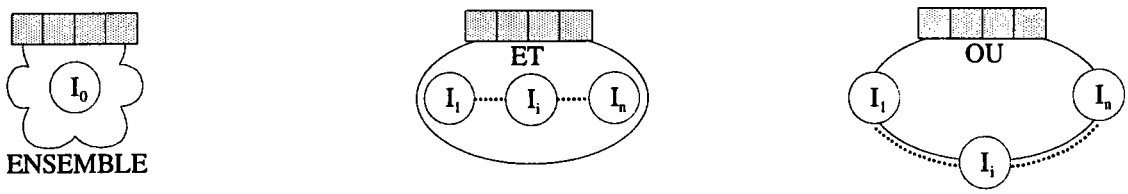


Figure III.5. *Les opérateurs de composition.*

Sur ces schémas, les interactions sont schématisées par une lettre I dans un cercle. Cet objet schématise l'interaction ou les interactions sur lesquelles s'applique l'opérateur. Cette notation est utilisée pour présenter les opérateurs de manière abstraite mais peut très bien être utilisée par l'outil graphique pour favoriser l'abstraction. De cette façon, les réseaux sont vus à différents niveaux, le passage entre les niveaux ne posant aucun problème : il suffit de "dérouler" ou "d'enrouler" les interactions.

5.2.2. Les tâches.

C'est une entité destinée à regrouper des actions ayant un objectif conceptuel commun. Elle est entièrement spécifiée par son nom et ses paramètres.

Figure III.6. *Une tâche.*

Selon le niveau d'abstraction que l'on désire, une tâche peut être représentée par :

- une action globale décrite avec le formalisme lui-même. Cette possibilité facilite la réutilisation en s'appuyant éventuellement sur des bibliothèques. Dans ce cas, la tâche possède un résultat si l'action globale en possède un. Remarquons que ce point de vue est important, il autorise des extensions récursives du formalisme comme dans le cas des réseaux de transitions;
- une action globale décrite par un programme NADRAG tel qu'il a été présenté. Dans ce cas, la tâche peut être interactive ou non. Si elle est interactive, on constate que l'on se trouve dans un cas très intéressant de contrôle mixte du dialogue, le formalisme et NADRAG se partagent les contrôles externe et interne. Comme dans le cas précédent, la tâche possède un résultat si le programme associé en possède un;
- une action non interactive (action atomique) écrite dans un langage conventionnel. C'est une action qui ne comporte aucun dialogue entre l'opérateur et le système; on peut rapprocher cette tâche d'un programme NADRAG ne comportant pas de dialogue. Une telle entité est généralement une interface vers une activité extérieure.

Notons qu'une tâche est exécutée seulement lorsque tous ses paramètres sont validés (interactions validées, ...).

5.3. Les relations.

Leur rôle est de modifier le fonctionnement d'une entité ou de spécifier celui d'une action globale. Concernant le fonctionnement d'une entité, on rencontre les contraintes et les effets, alors que pour l'action globale, on trouve la séquence et le flot de données. Nous présentons ces quatre relations dans les paragraphes suivants.

5.3.1. Les effets.

C'est un traitement effectué alors que le système attend une réponse de l'opérateur. L'effet peut être très varié et dépendre aussi bien de l'opérateur que du système. On peut l'associer à toute entité du formalisme.

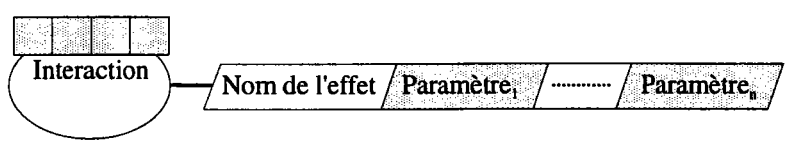


Figure III.7. *Un effet sur une interaction.*

Un ou plusieurs effets peuvent être associés à une entité pour cumuler les aides fournies à l'opérateur. On peut rencontrer des aides dynamiques pour aider l'opérateur à comprendre comment le système interprète ses intentions (affichage des objets tels qu'ils seraient construits si l'opérateur poursuit dans la voie qu'il a choisie, ...). On peut également avoir des aides pour comprendre ce que veut le système (mise en évidence d'objets, ...).

5.3.2. Les contraintes.

C'est une relation n-aire que l'on peut associer à toute entité ayant un résultat (objet, ensemble d'objets ou autres) : les interactions et les tâches avec résultat. Des contraintes liées à une interaction permettent de restreindre le domaine de validité des objets pouvant être désignés par l'opérateur.

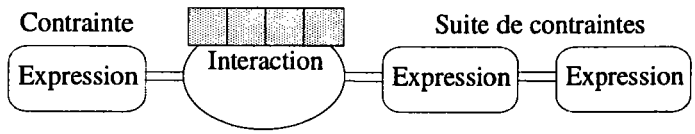


Figure III.8. *Une contraintes et une suite de contraintes.*

On appelle suite de contraintes, une liste de contraintes que l'objet concerné doit vérifier. Plusieurs suites de contraintes peuvent être associées à une entité pour obtenir des domaines de validité disjoints.

5.3.3. Le flot de données.

Cette relation indique d'où viennent et où vont les données. Une donnée est soit le résultat d'une interaction, soit le résultat d'une tâche. Un flot de données est principalement utilisé pour indiquer où vont les objets reçus par les différentes interactions et d'où viennent les paramètres des différents traitements telles que les contraintes, les tâches, ...

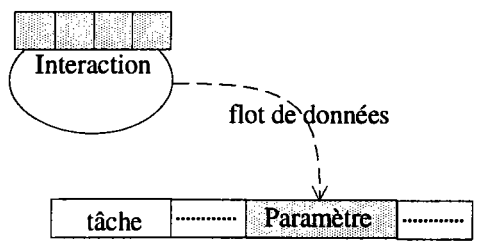


Figure III.9. *Un flot de données sur une interaction.*

À tout flot de données, on peut associer une fonction effectuant des modifications de la donnée (accès à la dernière réponse d'un ensemble, calcul de la surface d'un contour, ...). Grâce à cette relation, le programmeur d'interfaces ne se préoccupe pas de l'utilisation éventuelle de variables (tout se fait directement en pointant directement les objets ou en réalisant des copies de flots de données). On augmente ainsi la connaissance sémantique des objets : d'un simple coup d'œil, le programmeur d'interfaces repère les objets concernés par le traitement, ce qui n'est pas si aisé lorsque de nombreuses variables intermédiaires ont été utilisées.

5.3.4. La séquence.

La séquence indique de manière explicite la suite des opérations à effectuer : le contrôle passe d'entités en entités en suivant les relations de séquence (le début des opérations est indiqué par l'entité de départ).

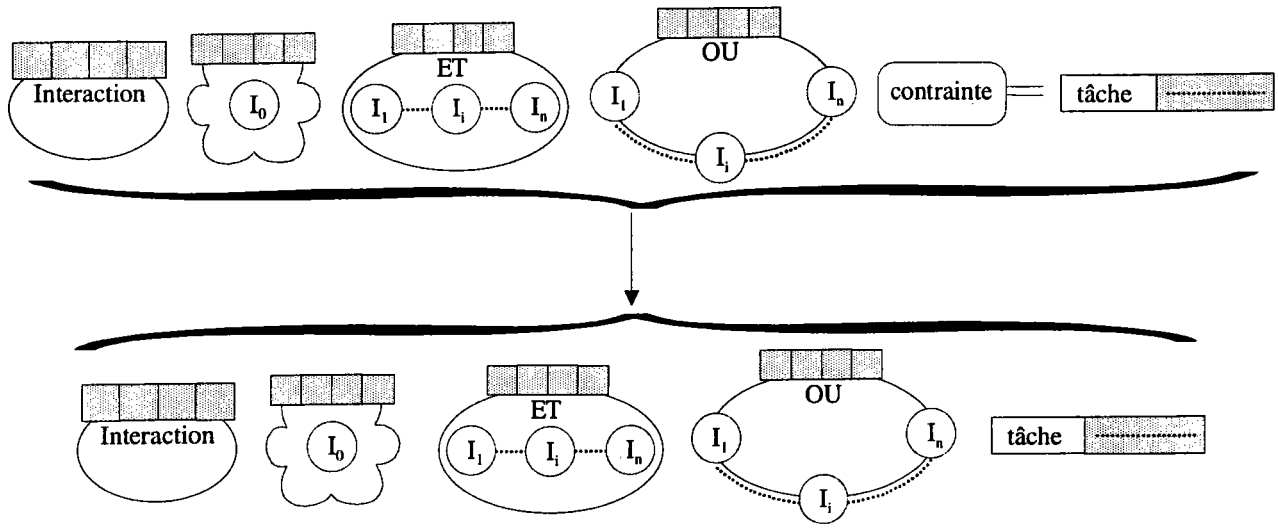


Figure III.10. *Une séquence (les entités de départ et les entités d'arrivée).*

Une séquence est fixée entre des entités de même niveau. En particulier, il est impossible de fixer une séquence à partir d'une interaction qui compose les opérateurs ET ou ENSEMBLE. Cela signifie que la relation séquence doit s'appliquer à l'opérateur dans sa globalité. Par contre, pour l'opérateur OU, il est bien entendu possible de préciser pour chaque interaction qui le compose, la séquence à suivre en cas de validation (les traitements peuvent varier selon l'interaction validée).

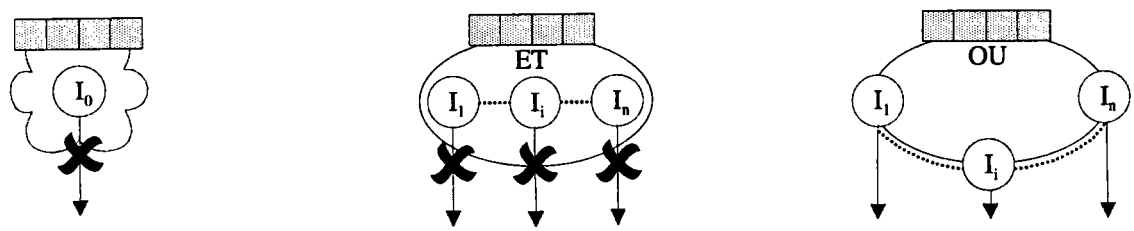


Figure III.11. *Les règles du séquencement.*

Remarquons que les schémas des opérateurs tiennent compte de ces règles : les interactions des opérateurs ET et ENSEMBLE sont à l'intérieur pour figurer une "barrière" infranchissable ou une hiérarchie alors que les interactions du OU sont au même niveau que l'opérateur (sur sa périphérie). C'est très important pour la compréhension rapide des manipulations possibles.

Avec la séquence, nous abordons le problème de l'ordonnancement des dialogues. Notre objectif est de fournir au programmeur d'interfaces les outils pour que la spécification soit la plus aisée et la plus rapide possible. Dans certains cas, l'opérateur doit exécuter un dialogue (des interactions) dans un ordre précis et pré-défini (cas du dialogue figé). Il exécute une série d'interactions en fonction du choix de l'agencement. La séquence est alors indiquée de manière explicite entre les entités du formalisme.

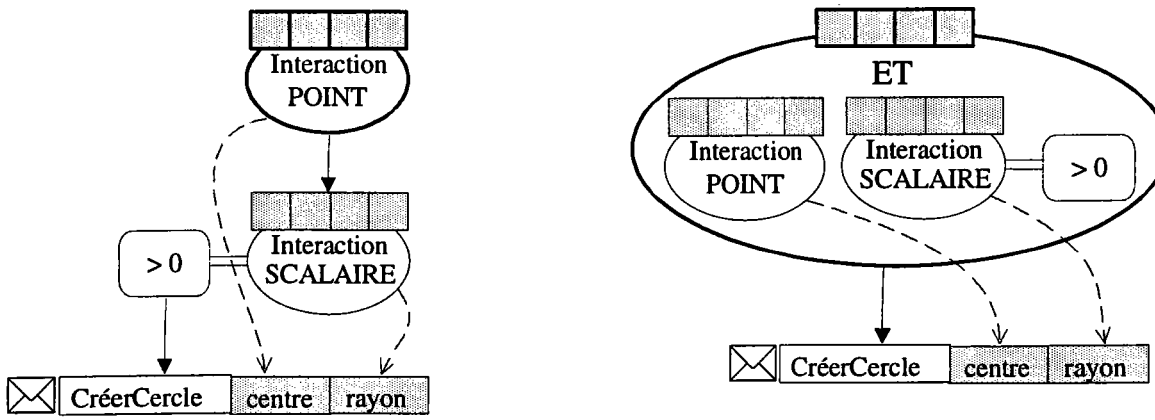


Figure III.12. *Mise en place d'une séquence explicite. Dans le premier cas, l'opérateur doit donner le centre puis le rayon. Dans le second, il a le choix de les donner dans n'importe quel ordre mais c'est le programmeur d'interfaces qui l'a déterminé.*

Dans la mesure où SACADO laisse un maximum de liberté à l'opérateur lors de l'utilisation du système, on est en droit d'attendre de même lors de la conception. Il est logique d'éviter la mise en place du séquencement des dialogues dès que cela est possible. En fait, l'ordre est fixé, non pas par le programmeur d'interfaces (sauf cas particuliers), mais par la présence de dépendances induites par les contraintes, les effets, ... Dans l'exemple précédent, le programmeur d'interfaces a indiqué explicitement un ordre pour l'obtention du centre et du rayon alors que les contraintes et les types des objets demandés suffisent amplement. Par la suite, nous appelons non-déterminisme le fait de ne pas indiquer explicitement la séquence d'actions. Cette notion est très importante pour l'outil graphique car la détermination de l'ordre des actions est à sa charge.

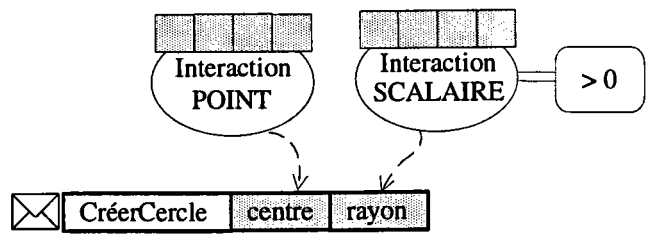


Figure III.13. *Mise en place d'une séquence implicite. Le programmeur d'interfaces n'a pas fixé l'ordonnancement des dialogues, le système détermine automatiquement que la solution la plus libre est la seconde de la figure III.12.*

La description d'une action globale se transforme dans la majorité des cas en une séquence explicite de tâches, chaque tâche possédant des paramètres liés aux interactions grâce aux flots de données. L'action déterministe est générée automatiquement par "simple" analyse des dépendances entre les interactions et leurs relations. On soulage le programmeur d'interfaces d'une tâche inutile et forcément fastidieuse dans la mesure où elle peut être déduite.

5.4. Les actions globales.

Nous allons montrer à partir de plusieurs exemples, la méthode de développement, à travers les étapes de construction d'une action globale, ainsi que l'introduction de chacune des notions du formalisme et

en montrer l'intérêt. Le premier exemple est une action globale de construction d'un segment défini par deux points distincts. Dans cette action globale, on distingue déjà trois éléments essentiels :

- la tâche proprement dite, de création du segment qui est non interactive;
- les deux points;
- la (ou les) contrainte(s) permettant d'obtenir des points distincts.

La première étape consiste à décrire ce que l'on veut faire afin de réaliser l'action désirée. Dans l'exemple, on désire simplement réaliser un segment, ce qui est fait par une tâche que l'on appelle CréerSegment2pts (cette tâche peut être un programme NADRAG comme décrit précédemment). Le résultat de cette action globale est le résultat de cette tâche, le segment lui-même.

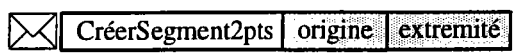


Figure III.14. *La tâche de l'action globale.*

La tâche apparaît en traits forts car elle indique où l'action globale débute (il n'y a aucune difficulté car l'action ne comporte qu'une seule tâche).

La seconde étape permet de satisfaire la réalisation (l'exécution) dans de bonnes conditions des différentes tâches; pour cela, il convient de fournir les paramètres de création du segment. Il faut mettre en place les interactions permettant la récupération des objets, à savoir les deux points. De plus, on indique clairement les liens existant entre les interactions et les paramètres ce qui évite d'utiliser des variables. Le type des interactions peut être déduit à partir de la spécification de la tâche issue des bibliothèques des actions globales et des actions non interactives. De même, les compatibilités sont connues (au moins en partie) grâce à la bibliothèque des interactions (seuls les cas particuliers restent à préciser).

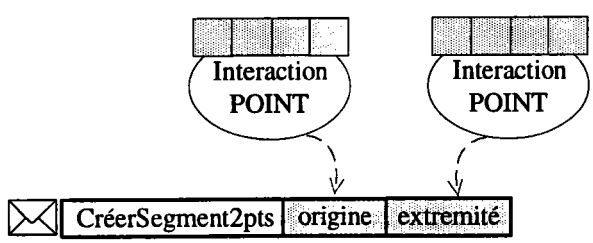


Figure III.15. *Le paramétrage d'une tâche.*

On peut remarquer qu'à ce stade de la spécification, le schéma peut être considéré comme un prototype simple de l'action globale finale désirée.

La troisième étape doit permettre d'affiner la spécification en apportant les conditions portant sur les différentes interactions. Pour cela, on utilisera des contraintes afin d'indiquer les domaines de validité des différents objets. On obtient finalement le schéma suivant :

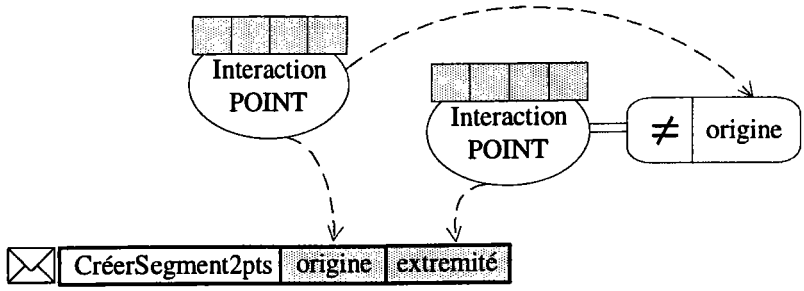


Figure III.16 *Les contraintes de validité.*

Dans cet exemple, la contrainte de différence est une contrainte binaire mais on peut également avoir des contraintes un-aires ne portant que sur une interaction. Il faut également noter qu'à partir de cette spécification, l'outil peut maintenant fixer l'ordre des dialogues : l'opérateur doit donner un point, puis un second point différent du premier, avant de créer le segment dans de bonnes conditions. Le programmeur d'interfaces est déchargé de cette opération et le système peut faire évoluer l'action globale au fur et à mesure des relations introduites. On décrit les traitements à effectuer, les éventuels paramètres (objets) fournis par l'opérateur, ainsi que les contraintes agissant sur ces objets. La construction devient alors un prototype de bien meilleure qualité.

La quatrième étape de la conception porte sur la très forte interactivité nécessaire dans les systèmes de C.F.A.O. Chaque action de l'opérateur doit faire l'objet d'un écho par le système afin de montrer qu'il a correctement interprété cette action. Il faut alors associer des effets aux interactions. Nous appliquons cette étape en précisant que l'on désire avoir pour chaque valeur de l'extrémité, une esquisse du segment.

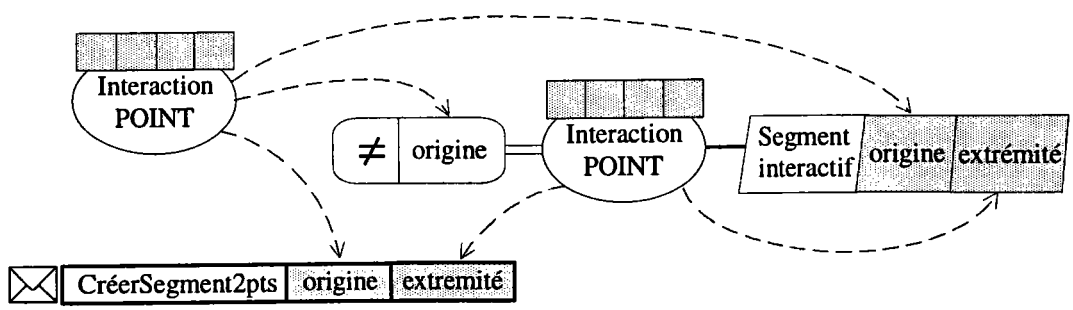


Figure III.17 *Un effet interactif.*

La compréhension d'un tel formalisme est très rapide par rapport à un programme similaire écrit à l'aide de NADRAG. Les entités graphiques et les relations facilitent grandement l'analyse d'une action globale. Il ne faut toutefois pas perdre de vue que ce formalisme ne possède pas en l'état la puissance de description de NADRAG.

5.5. Implantation et conclusion.

Les travaux réalisés autour de ce formalisme sont nombreux. On peut citer notamment la modification de la base de développement de SACADO ainsi que deux maquettes concernant l'outil graphique de conception.

Le noyau de fonctionnement de SACADO a été modifié pour la prise en compte des actions globales décrites selon un schéma déterministe simplifié. L'annexe C détaille les modifications apportées et les avantages que l'on a tirés de cette évolution.

Une première implantation de l'outil graphique a été réalisée dans le cadre d'un stage de D.E.A. [LAH 95]. Cette maquette a permis de valider les concepts du formalisme et d'évaluer sa facilité d'utilisation en confrontant des points de vue différents. Nous avons développé une seconde implantation comme exemple des propositions faites dans le chapitre 4. La description de cette maquette se trouve dans l'annexe D.

Comme nous l'avons montré à travers les exemples d'utilisation, ce formalisme autorise une grande flexibilité notamment par les informations implicites (variables, séquence des dialogues) et par la possibilité de prototyper les actions globales selon des niveaux différents d'abstraction. Les opérateurs de systèmes de C.F.A.O. ont l'habitude de manipuler des données graphiques et cette approche par un formalisme graphique les maintient dans ce cadre. La construction des actions globales est ainsi plus proche d'un jeu de LEGO® que de la programmation. Son intérêt réside également dans la rapidité de compréhension et d'assimilation des actions globales décrites. Les modifications sont simples et rapides à mettre en œuvre. À ce titre, il nous sert de base pour la génération du dialogue que nous proposons dans le chapitre 4. Les actions globales sont générées sous une forme accessible au programmeur d'interfaces avec des outils de manipulation et de modification adaptés.

6. CONCLUSION.

Tout au long de ce chapitre, nous nous sommes attachés à présenter les outils de développement mis à la disposition du programmeur d'interfaces pour utiliser le modèle de dialogue de SACADO. Avec ces outils, SACADO devient un véritable système de gestion d'interfaces utilisateur (SGIU); il dispose d'un modèle d'interaction, de deux modèles de dialogue et des outils pour spécifier les différents composants.

Au début nous nous sommes intéressés à l'aspect statique du dialogue : les menus, les compatibilités entre les menus et les associations menus-actions globales. Mais cet aspect n'est pas suffisant pour la gestion complète de l'architecture du dialogue et nous sommes donc allés plus loin. L'action globale ne peut pas être considérée comme un tout indissociable et nous sommes descendus au niveau de l'élément unitaire du dialogue pour autoriser la plus grande liberté possible de spécification. Cette décomposition nous a confortés dans notre vision, que nous qualifions de naturelle, car en fin de compte c'est le dialogue et sa primitive INTERACTION, qui nous semble le plus adapté à la définition de l'architecture d'un logiciel interactif.

Nous avons donc proposé deux langages d'actions pour la définition des actions globales du système : un langage textuel, NADRAG, et un langage graphique basé sur un formalisme.

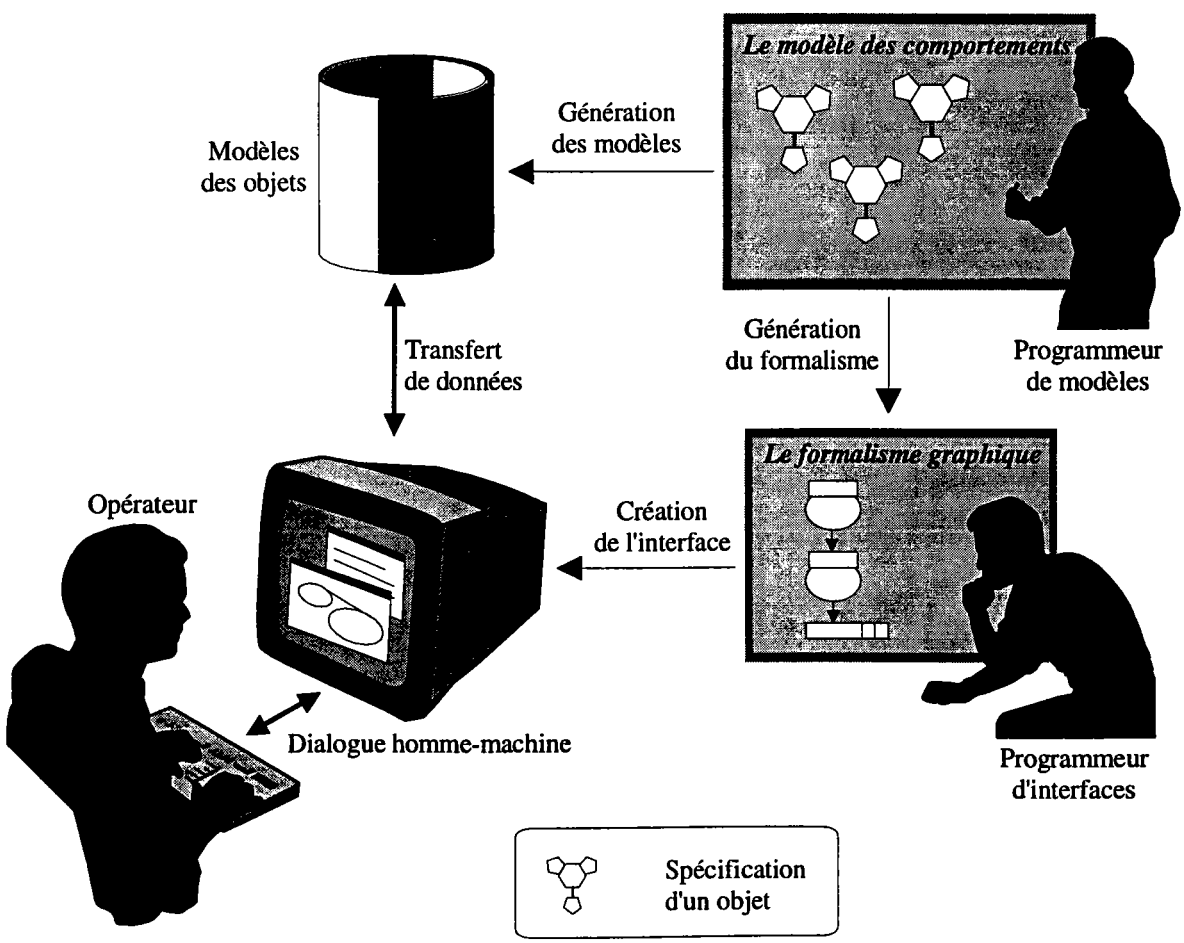
Tout en possédant les caractéristiques fondamentales d'un langage classique, NADRAG est simple à utiliser car il est destiné à des concepteurs de bureaux d'études non-informaticiens mais qui sont des utilisateurs spécialisés. Sa capacité d'évolution est assurée par la modularité de sa réalisation. Ainsi, il peut très bien rapprocher le développement du dialogue et de l'application, rapprocher le prototype et le produit final. Le système se trouve alors programmé à un seul et même niveau par tous les intervenants.

Le langage graphique que nous avons spécifié est un langage visuel primitif qui laisse le programmeur d'interfaces construire et modifier interactivement une application. L'outil graphique associé se base sur les connexions mises en place entre les éléments du dialogue (entités du formalisme). Une action globale construite avec ce formalisme est ensuite générée sous forme NADRAG pour être intégrée dans l'implantation SACADO. La description se fait donc selon deux niveaux : le programmeur d'interfaces utilise un langage visuel alors que le programmeur d'applications, plus spécialisé, peut utiliser un langage plus conventionnel et en particulier NADRAG qui est totalement intégré dans le formalisme en tant que tâche.

Ces deux langages d'actions accessibles aux trois utilisateurs de SACADO simplifient le processus de construction des applications de C.F.A.O. en offrant la possibilité d'impliquer des non-programmeurs dans la conception et la réalisation. Ils profitent au maximum des solutions apportées par notre modèle d'interaction qui a permis la modélisation de l'action globale autour du dialogue et de la primitive INTERACTION. Plusieurs implantations et tests ont montré l'utilité de ces langages, mais la simplicité et la complexité de certaines actions globales tendent à prouver que le système doit aller plus loin dans l'aide apportée au programmeur d'interfaces. Dans le cas des actions simples, le système peut les déduire et pour les actions plus complexes, il doit fournir un squelette ou un cadre de travail. Pour répondre à cette demande, nous présentons dans le chapitre suivant, une extension du modèle générique dans lequel sont décrites des informations, les comportements, qui servent entre autres à la génération du dialogue.

CHAPITRE 4.

LES MODÈLES ET LA GÉNÉRATION DU DIALOGUE.



1. INTRODUCTION.

Comme nous l'avons étudié dans les chapitres précédents, le dialogue homme-machine est d'une importance capitale pour la plupart des applications informatiques. La C.F.A.O. prise dans sa signification la plus large, comporte un certain nombre de spécificités : modèles à structures complexes, données variées, très grand nombre d'informations dans un modèle donné, variété des comportements et des vues externes (variations de formes, types de vues ...) ... On tend ainsi vers un univers virtuel, dans lequel l'opérateur humain doit disposer d'outils conviviaux, pour définir et modifier des objets, et de possibilités de navigation. Cependant, alors même que les modèles utilisés, même s'ils sont souvent mal formalisés, deviennent de plus en plus complexes (gestion des contraintes, formes quelconques ...), les programmeurs de systèmes continuent à construire les interfaces homme-machine de façon totalement expérimentale : le seul progrès notable est l'utilisation (parfois à tort et à travers) d'outils tels que les widgets et les présentations standard [JOH 93; MYE 91].

Pourtant, la connaissance étant répartie entre les utilisateurs du système et les modèles, il paraît judicieux de tenter d'utiliser au mieux cette complémentarité pour déduire un dialogue adaptatif : le terme adaptatif signifie que ce dialogue doit pouvoir être adapté aux objets et contraintes modélisés et au type d'utilisateur.

2. LE DÉVELOPPEMENT DU DIALOGUE.

Un certain nombre d'aides générales peuvent être apportées à l'opérateur. Ces aides, essentiellement visuelles, sont particulièrement mises en évidence lorsque l'on étudie les outils de navigation. Citons pour mémoire les aides visuelles (attractions, navigateurs intelligents, ...) et les didacticiels permettant de comprendre les concepts du système de C.F.A.O. Mais, on peut envisager les aides à la définition du dialogue du point de vue du programmeur d'interfaces. En effet, celui-ci définit l'ensemble des interactions et leur séquencement. Pour l'aider, nous avons implémenté plusieurs approches basées sur l'utilisation de générateurs. Nous distinguons deux classes de générateurs :

- les générateurs applicatifs;
- les générateurs primaires.

Les générateurs applicatifs ont pour but de créer une application qui utilise une implantation SACADO [GAR 90]. Ils permettent la définition complète du modèle, du dialogue et des actions de l'application, avec la création de nouveaux types d'actions sans connaissance, a priori, d'un schéma de mise en œuvre. NADRAG est un exemple de ce type de générateur. Cette approche a montré son intérêt, mais elle impose une discipline importante au programmeur d'interfaces dans la définition des actions. Cette discipline n'est pas obligatoirement contradictoire avec une bonne utilisation de la C.F.A.O., mais nous avons cherché à automatiser un maximum de cas. Par exemple, pour éviter une trop grande lourdeur de définition, nous avons implémenté des générateurs primaires et des bibliothèques d'interactions.

Les générateurs primaires permettent de compléter les actions d'une application implantée sous SACADO. Le schéma de l'action est parfaitement défini (dialogues et traitements). Un exemple est donné dans [KOH 91] avec le générateur de constructions sous contraintes. Ces actions de construction peuvent se décrire sous une forme générale qui les définit complètement et seul un paramétrage est nécessaire au générateur pour réaliser l'action correspondante.

Dans ce chapitre, notre objectif est d'apporter une aide plus importante au programmeur d'interfaces en nous basant sur les connaissances que possède le modèle (des objets) et en rapprochant les deux générateurs de base (générateur de modèles, générateur de dialogue et d'architecture). À partir de la définition de la primitive unique de dialogue INTERACTION, qui comprend l'ensemble des éléments qui nous paraissent indispensables lors de toute interaction opérateur-système, nous tentons d'apporter une réponse au problème suivant : dans quelle mesure peut-on déduire les actions et leur séquencement à partir de la connaissance du modèle ? Il s'agit du problème inverse de celui que nous avions résolu dans la maquette SACADO.

Comme dans de nombreux domaines, mais sans doute de manière souvent plus critique, la mise en œuvre d'un système de C.F.A.O. implique un dialogue entre les différents intervenants pour aboutir à un ensemble opérateur-système cohérent. En général, on peut aboutir à deux défauts majeurs dans la définition de ce dialogue :

- il n'est pas complet : le programmeur d'interfaces, essentiellement soucieux de mettre en place un dialogue adapté à l'opérateur, peut omettre des possibilités inhérentes au modèle. Par exemple, les connaissances décrites dans le modèle peuvent par association de contraintes créer des objets d'une manière difficilement accessible au programmeur d'interfaces;
- il propose des fonctions que le système ne peut pas traiter : le programmeur d'interfaces, répondant à une demande d'un opérateur peut créer un dialogue qui ne peut pas être pris en compte par le système, soit parce qu'elle nécessite des objets non décrits, soit parce qu'un traitement indispensable n'a pas été programmé.

Nous nous attachons dans ce chapitre à la définition de l'interface homme-machine, de telle sorte qu'elle tienne compte de l'ensemble des possibilités modélisées et qu'elle puisse être adaptée à l'opérateur. Notre approche doit donc prendre en compte les connaissances qui ont été introduites dans le modèle (par exemple, un cercle est connu par son centre et son rayon et peut être construit par tangence ou par des points de passage ...). De plus, elle doit également tenir compte du fait que l'opérateur ou le concepteur d'interfaces dispose d'un certain nombre de connaissances non modélisées dans le système de C.F.A.O. (car elles sont trop complexes ou spécifiques à un opérateur) et qu'il a une vision plus synthétique de l'apparence du dialogue que le système.

Pour aller plus loin dans cette utilisation du modèle comme base de dialogue, nous envisageons de fournir des aides de différentes formes :

- préciser le fonctionnement d'une interaction donnée : considérons, par exemple, une interaction demandant à l'opérateur de placer une vis. Si le modèle comprend un certain nombre d'informations sur les contraintes, on pourra en déduire que ce placement ne peut se faire que dans un trou et, sans doute, en déduire les trous (de diamètre adéquat) acceptables dans le modèle. Cette analyse peut se traduire par une aide visuelle (mise en évidence des trous candidats) et éviter ainsi des actions inutiles, voire sans intérêt. La notion de menu local peut permettre à l'opérateur de créer un trou adéquat s'il le souhaite, sans que le programmeur d'interfaces n'ait à définir d'interaction supplémentaire. Le traitement des contraintes, plutôt que de se faire directement dans le modèle se fera de manière plus performante dans un modèle applicatif, correspondant à un découpage du modèle (image du modèle selon la vue utilisée);
- autoriser des objectifs "flous" : il est possible d'imaginer des interactions dont les objectifs (par exemple en identification) sont imprécis. Les connaissances du modèle permettent alors de gérer des notions d'anticipations [ALP 93; GRA 94]. Si l'on est par exemple en phase de création d'un segment, on activera les différentes contraintes possibles à l'approche d'un objet quelconque (tangent à, passant par ...);
- créer des actions automatiquement : on comprend dans des cas simples que l'on peut complètement définir des actions interactives pour la création d'objets [GIE 92]. Par exemple, si le modèle sait qu'un segment peut être créé comme "passant par deux points", il est clair que l'on peut en déduire l'action composée :

  Interaction (P_1); {point demandé; menu POINT local}
  Interaction (P_2, $P_2 \neq P_1$); {point demandé; menu POINT local}
  Appel à la fonction de création de segment du modèle;

  De telles créations peuvent être largement facilitées par l'utilisation des différentes bibliothèques d'actions;
- expliquer le fonctionnement d'une interaction : à partir de la connaissance du modèle, on peut donner une explication à l'opérateur sur le fonctionnement de l'interaction, voire de l'action globale. Cette explication peut se faire sous la forme d'un exemple type, de préférence graphique [LEM 92; SZE 94].

Il s'agit donc d'exploiter les connaissances décrites dans le modèle afin de déduire un dialogue dont la qualité essentielle est de proposer toutes les possibilités de l'application [FOL 93]. On peut montrer que certaines actions sont relativement faciles à déduire, mais que d'autres se révèlent très complexes. Ce dialogue peut être présenté en respectant le langage graphique introduit dans le chapitre précédent. Notre proposition est de mettre en place un outil profitant au mieux des connaissances du modèle et de l'expérience du programmeur d'interfaces.

Dans un premier temps, nous abordons la méthodologie que nous préconisons pour la spécification du modèle, en nous intéressant exclusivement aux composants qui ont une influence sur l'opérateur et son dialogue. Dans un second temps, nous montrons les deux approches complémentaires qui sont la base de notre travail : la déduction d'un dialogue à partir des connaissances du modèle et son adaptation par l'opérateur.

3. LE MODÈLE ET LES COMPORTEMENTS.

Il est habituel de présenter les modèles utilisés en C.F.A.O. sous la forme d'un ensemble de données : modèle B-rep ou CSG pour la géométrie ou modèles de caractéristiques pour les informations non géométriques.

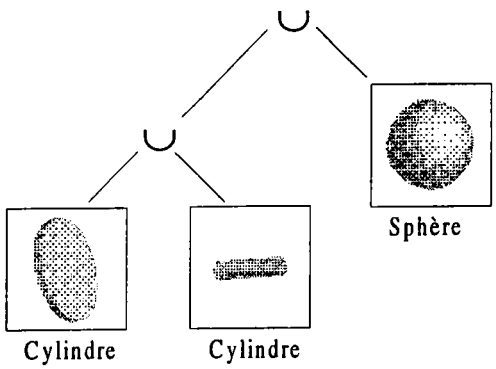


Figure IV.1. *Une modélisation CSG (l'objet est une combinaison d'objets primitifs).*

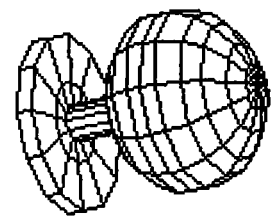


Figure IV.2. *Une modélisation B-REP (l'objet est représenté par sa "peau").*

Pourtant à ces modèles descriptifs s'ajoutent des modèles de fonctionnement (ou de comportement) qui intègrent une connaissance sur la création et la manipulation des modèles descriptifs [GAR 91]. C'est l'ensemble des modèles descriptifs et de fonctionnement qui nous paraît la meilleure base pour valider l'approche de déduction du dialogue à partir du modèle.

En effet, l'intégration du modèle de comportements dans la modélisation permet de concentrer toute la sémantique de l'objet dans l'objet. Cette sémantique est décrite une et une seule fois et tous les modules s'appuient dessus. De cette façon, elle ne se trouve pas dispersée sur plusieurs couches logicielles et les risques de sur-spécifications ou d'incohérences se trouvent réduits. Elle garantit la validité des dialogues de création et de manipulation déduits. Nous parlons de sémantique interne.

Nous considérons donc un objet comme un couple données-comportements que l'on pourrait assimiler au couple données-méthodes des langages orientés objets (LOO). Les comportements sont séparés en trois familles [GAR 95] :

- les *producteurs*, qui sont des comportements inhérents à la création des objets. Ils permettent de préciser un objet par des définitions et des informations de création;
- les *réacteurs*, qui caractérisent les objets. Nous parlerons de réacteurs extrinsèques pour préciser l'apparence des objets et de réacteurs intrinsèques pour préciser leur sémantique;
- les *transmuteurs*, qui concernent la modification des objets. Les transmuteurs extrinsèques changent le type des objets alors que les transmuteurs intrinsèques n'en modifient que l'image (il s'agit de réaliser une transformation temporaire de l'objet).

Notre propos se limite à supposer que le comportement est défini pour chaque objet. Un comportement plus général peut toujours être considéré, au moins en première approche, en le dupliquant pour chaque objet concerné.

3.1. Les objets.

Pour le système, un objet est défini par son modèle (représentation informatique). Comme nous l'avons vu précédemment, un objet est entièrement modélisé par un couple données-comportements. Les données définissent la géométrie et la topologie de l'objet. Par suite, les informations stockées dans les données ne peuvent correspondre à une simple énumération des attributs caractéristiques de l'objet. Nous distinguons deux parties :

- la représentation;
- les propriétés.

La représentation d'un objet contient la définition de ses attributs et leurs conditions de validité. De manière générale, un objet possède un nombre limité de représentations mais il possède une représentation canonique à laquelle les autres représentations peuvent se ramener. Cette représentation canonique est la représentation de l'objet, elle le définit entièrement.

Les propriétés sont considérées comme des objets à part entière et sont décrites ultérieurement par l'intermédiaire de l'utilisation des réacteurs. Nous avons choisi de les décrire au même niveau que les objets car elles sont au moins aussi importantes qu'eux. Dans de nombreux cas, l'opérateur désire satisfaire une propriété avant même de manipuler les objets. De plus, nous souhaitons obtenir une définition générique des propriétés par leur sémantique propre et ce, indépendamment des objets bien sûr. Cela impose de séparer les objets des propriétés.

La représentation canonique constitue une passerelle entre les différents comportements. Pour un objet de type α, on parle du domaine D_α de définition. Nous le notons comme le produit cartésien de domaines ($TYPE_i$), chaque domaine étant nommé (A_i) dès lors qu'une condition doit y faire référence. À tout instant, la représentation doit fournir une définition cohérente de l'objet.

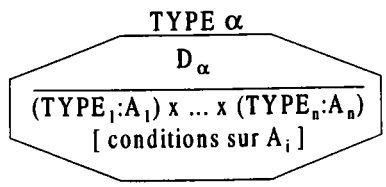


Figure IV.3. *Un objet de type α.*

Comme exemple de définition, nous proposons les objets CERCLES et SEGMENTS. Les objets sont simplement définis par leur représentation (domaine de définition).

Figure IV.4. *Les objets CERCLES et SEGMENTS.*

Pour définir entièrement le modèle des objets, nous introduisons l'application OBJ dont le rôle est de mémoriser les objets du modèle et que l'on définit par :

$$OBJ(\alpha) = \{ \text{ ensemble des objets de type } \alpha \text{ créés dans le modèle } \}$$

Toutefois, au delà de sa représentation, un objet est perçu par l'opérateur grâce aux différents modes de création. Pour cela, nous proposons de spécifier les créations parallèlement à la représentation de l'objet. Les comportements chargés de cette description sont les producteurs.

3.2. Les producteurs.

Les producteurs permettent de préciser un objet par les différentes manières de le créer : outre la définition de l'objet, ils doivent fixer son domaine de validité.

Un producteur P d'un objet de type α est défini par :

- le domaine de création Dc;
- la fonction de production Fp.

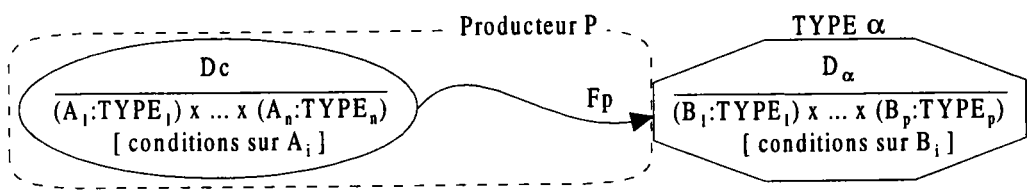


Figure IV.5. *Définition d'un producteur.*

Le domaine de création représente le domaine de validité des objets que l'on produit grâce au producteur concerné. Il s'apparente au domaine des objets, la différence provenant du fait que le domaine d'un producteur est associé à une représentation alors que le domaine de l'objet a un rôle fédérateur (tous les objets de même type ont la même représentation).

Un producteur évolue tout au long des interactions avec l'opérateur. Nous appelons état du producteur la connaissance de ce qui est défini et de ce qui reste à définir pour réaliser complètement un objet avec un producteur. La fonction de production Fp dépend donc de cet état. Il est possible pour chaque état du producteur de définir une fonction de production différente pour construire l'objet de la manière la plus précise possible.

$$\begin{array}{lllll} Fp & : & Dc \times \text{état} & \longrightarrow & D_\alpha \\ & & (o \ , \ e) & \longmapsto & Fp\ (o,e) \end{array}$$

Cette fonction se charge de la traduction du domaine du producteur vers le domaine de l'objet et ce, au fur et à mesure de son avancée. En particulier, lorsque le producteur est terminé (dans un état final), Fp est la fonction de création de l'objet final. Dans ce cas, on a :

$$OBJ(\alpha) = OBJ(\alpha) + Fp(o, \text{état final}) \quad \text{(l'objet final est ajouté dans le modèle)}$$

Comme nous l'avons vu dans la définition, un objet possède un nombre limité de représentations et une représentation canonique à laquelle les autres représentations peuvent se ramener. Ainsi, chaque objet peut contenir plusieurs producteurs, un producteur par représentation par exemple. Cela autorise des créations multiples et variées. L'exemple que nous avons choisi concerne l'objet CERCLES dont la représentation canonique est son centre et son rayon. Autour de ces données, s'articulent trois producteurs différents. Le premier, *CercleCentreRayon*, est directement lié à la représentation. Le deuxième, *Cercle3Pts*, impose la construction du cercle à partir de trois points. Le troisième, *CercleCentrePtdePassage*, impose la construction du cercle à partir d'un centre et d'un point appartenant au cercle. Tous ces producteurs contiennent les conditions de validité des attributs ainsi que la conversion vers la représentation canonique utilisée (les fonctions F_1, F_2 et F_3 de l'exemple sont définies uniquement pour l'état final des producteurs, lorsque tous les attributs sont connus).

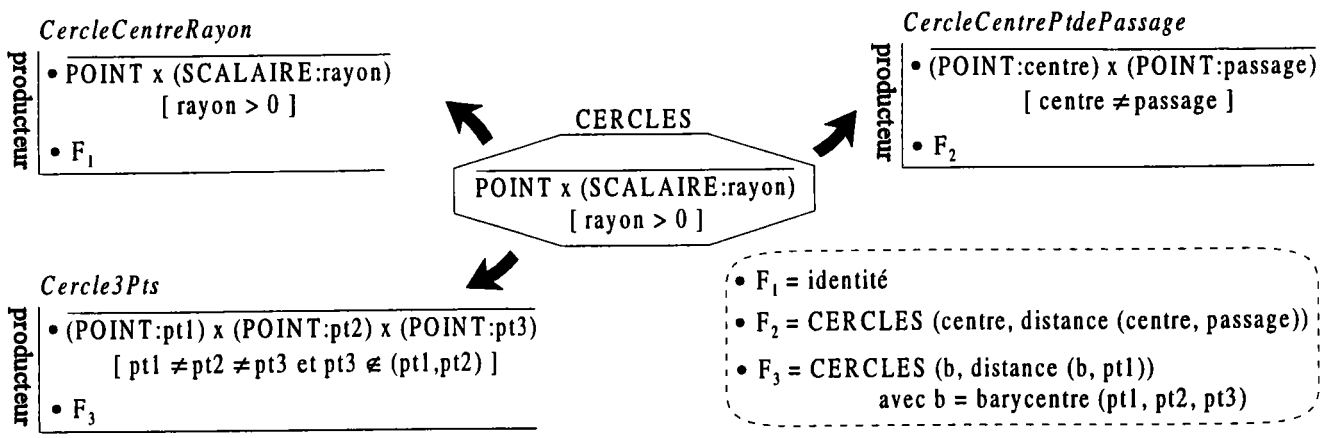


Figure IV.6. *Définition de CERCLES et de trois producteurs.*

Les producteurs expriment la sémantique des objets. Par exemple, le producteur *Cercle3Pts* décrit une façon de construire un cercle. Pour cela, il s'appuie sur le fait qu'il n'existe qu'un seul cercle passant par trois points distincts et non alignés. C'est une façon de représenter la sémantique des cercles. Il est important d'intégrer ces producteurs à l'objet lui-même afin de les partager entre les différents applicatifs et de capturer les connaissances du programmeur de modèles.

Cette sémantique se retrouve encore dans l'ensemble défini par l'application OBJ :

$$OBJ(P) = \{ \text{ objets valides construits en fonction de l'état du producteur P } \}$$

En fonction de l'avancée des interactions, cet ensemble définit les objets que le producteur peut créer à partir des objets fournis par l'opérateur. Lorsque l'opérateur désire construire un objet à l'aide d'un producteur P, cet ensemble est l'ensemble des objets valides qu'il peut créer à un instant donné. Dans

l'exemple de l'objet CERCLES, nous considérons le producteur *CercleCentrePtDePassage*. Lorsque les deux points (p1 et p2) ont été fournis par l'utilisateur final, le producteur possède deux objets valides : (p1, p2) et (p2, p1). Cela renforce notre concept d'interaction standard : il suffit de mettre en place un style d'interaction particulier pour choisir la représentation désirée par l'opérateur.

3.3. Les transmuteurs.

Un objet est défini par rapport aux opérations que l'on peut lui appliquer, opérations qui dépendent essentiellement de son type. Ainsi, la détermination du type d'un objet après modifications prend toute son importance. Les transmuteurs sont les comportements chargés de modifier le type d'un objet en accord avec sa sémantique et les opérations que l'on désire réaliser.

De manière générale, le transmuteur s'occupe de tout changement de type en accord avec la sémantique de l'objet. Il diffère du producteur car il transforme un objet déjà créé. Un transmuteur T d'un objet de type α vers un objet de type β ($\beta \neq \alpha$) est défini par :

- le domaine de transmutation Dt;
- la fonction de transmutation Ft.

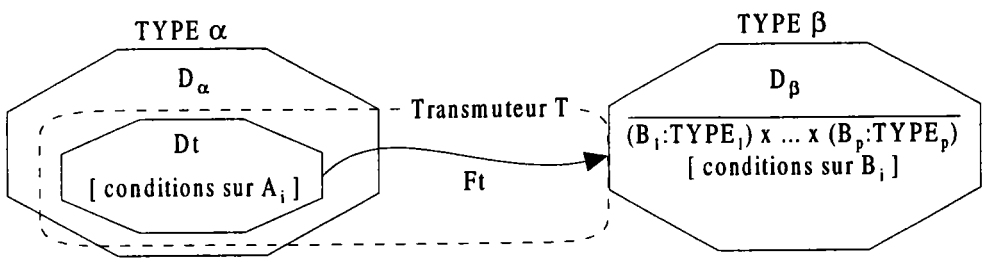


Figure IV.7. *Définition d'un transmuteur.*

Le domaine Dt est une restriction du domaine intial des objets D_α ($Dt \subset D_\alpha$). Il exprime les conditions que la transmutation doit vérifier et peut parfois être le domaine initial lui-même (dans ce cas, tous les objets peuvent être transmutés selon les besoins). Par exemple, seule une ellipse dont les deux foyers sont égaux se transmute en un cercle. Par contre, dans tous les cas, un cercle peut se transmuter en une ellipse.

La fonction de transmutation Ft est la fonction qui définit comment doit être construit l'objet transmuté à partir de l'objet initial.

$$\begin{array}{lll} Ft : & Dt & \longrightarrow D\beta \\ & o & \longmapsto Ft\,(o) \end{array}$$

Nous distinguons deux familles de transmuteurs :

- les transmuteurs intrinsèques (ils changent le type d'un objet);
- les transmuteurs extrinsèques (ils réalisent une transformation virtuelle d'un objet).

Nous détaillons ces deux types de transmuteurs dans les paragraphes suivants en insistant plus particulièrement sur les transmuteurs extrinsèques.

3.3.1. Les transmuteurs intrinsèques.

Ils modifient l'image d'un objet en fonction des traitements à lui appliquer. Il s'agit du concept de localisation / globalisation que nous avons abordé dans le chapitre 2 et qui permet une adéquation entre les objets et les traitements. C'est une transformation virtuelle de l'objet en fonction du contexte et des besoins. Lorsqu'un transmuteur intrinsèque T est utilisé pour transmuter un objet o de type α en un objet de type β, nous pouvons décrire cette transmutation de la manière suivante (en termes d'objets) :

$$OBJ(\alpha) = OBJ(\alpha) \qquad \text{(l'objet est conservé)}$$
$$OBJ(\beta) = OBJ(\beta) + Ft(o) \qquad \text{(la copie virtuelle est ajoutée dans le modèle)}$$

Les transmuteurs intrinsèques autorisent une représentation différente du modèle initial (modèle opérateur) pour réaliser des opérations particulières; on parle de modèle local. Une opération sur un objet peut s'avérer incompatible avec cet objet. Avant de la rejeter, le système tente de transmuter l'objet en un objet qui accepte cette opération. Deux cas peuvent se produire :

- le système connaît le transmuteur à appliquer. Dans ce cas, un transmuteur a été associé de manière explicite à l'opération et il n'y a pas de difficulté. Par exemple, le calcul de l'appartenance d'un point à un segment peut être associé à un transmuteur de segment en droite. Avant le calcul, le segment est transmuté en droite et le résultat de l'opération est alors l'appartenance du point au support du segment;
- le système ne connaît pas le transmuteur. Aucun transmuteur n'a été indiqué, il faut donc rechercher parmi tous les transmuteurs ceux qui donnent des objets acceptant l'opération. Dans ce cas, des ambiguïtés peuvent apparaître. Il peut y avoir plusieurs transmuteurs possibles et il faut indiquer explicitement lequel utiliser (des conventions ou des priorités peuvent également être mises en place).

En définitive, un modèle local est une représentation différente du modèle opérateur afin de permettre ou de faciliter des opérations particulières. Ce modèle est chargé de présenter les objets de manière adaptée aux algorithmes, en particulier aux algorithmes associés au dialogue. Un exemple de transmuteur intrinsèque est donné par la figure suivante.

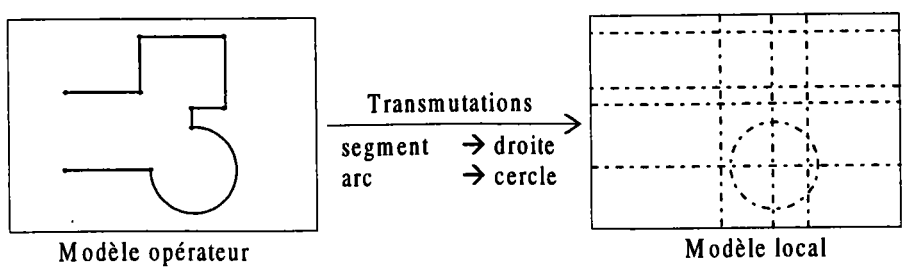


Figure IV.8. *Des transmuteurs intrinsèques.*

Dans cet exemple, nous désirons détecter l'appartenance d'un point aux supports des objets du modèle opérateur. Il suffit de décrire des transmuteur intrinsèques qui transmutent les segments en droites et les arcs en cercles. Ce transmuteur est ensuite associé à l'opération d'appartenance et on obtient le modèle local associé qui ne contient que les droites et les cercles supports des objets du modèle opérateur. Au cas où ce modèle local serait interne au dialogue, il permet un fonctionnement entièrement autonome et les échanges entre le dialogue et l'application sont diminués. De cette manière, l'interactivité se trouve facilitée et le filtrage des objets à prendre en compte se fait directement par la transmutation.

Ces transmuteurs mettent en évidence l'importance des modèles locaux dans la réduction des informations en mouvement entre le dialogue et le modèle. La réduction de la circulation est bénéfique pour une meilleure interactivité mais cela impose le maintien d'un grand volume d'informations. Les transmuteurs intrinsèques sont sans doute une solution pour décrire ce que l'on désire, mais ils font appel à des notions qui méritent une étude très approfondie. Il s'agit d'un problème fondamental de la multi-modélisation [GAR 91].

3.3.2. Les transmuteurs extrinsèques.

Ils décrivent une transformation réelle d'un objet, un changement de type. Il est alors possible d'utiliser les opérations propres à ce nouveau type d'objet. Lorsqu'un transmuteur extrinsèque T est utilisé pour transmuter un objet o de type α en un objet de type β, nous pouvons décrire cette transmutation de la manière suivante (en termes d'objets) :

OBJ (α) = OBJ (α) - o (l'objet n'est pas conservé)
OBJ (β) = OBJ (β) + Ft (o) (l'objet transmuté est ajouté dans le modèle)

Ces comportements éclairent une interaction floue en indiquant ce que va devenir l'objet manipulé. Dans le cas où l'opérateur répond à une interaction par un objet non attendu par le système, l'analyse de ses transmuteurs permet de poursuivre le dialogue si cet objet peut se transmuter en un objet valide.

Les transmuteurs permettent de préciser, d'adapter des objets en fonction du contexte. L'exemple que nous présentons est celui de la mise en place d'un arrondi sur un contour déjà formé. Pour réaliser cette opération, l'opérateur utilise un arc; le système se charge de le transformer en arrondi sous certaines conditions. Cette manière de faire, transformer un objet géométrique en une caractéristique, donne une fonction à un objet dans un certain contexte. Il n'y a plus d'ambiguïté entre arc et arrondi puisque le passage de l'un à l'autre se fait sous contrôle.

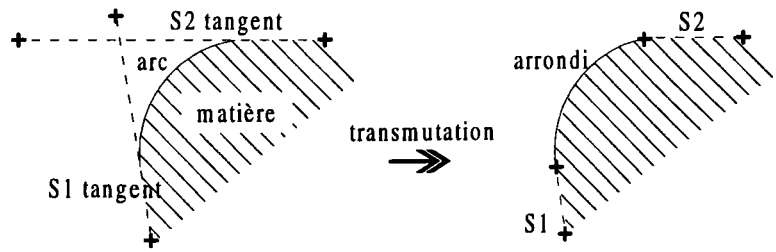


Figure IV.9. *Passage d'un objet géométrique à une caractéristique.*

Le déclenchement d'un transmuteur dépend de son utilisation et de son rôle. Nous distinguons essentiellement deux types de déclenchement :

- implicite;
- explicite.

Un déclenchement implicite est à la charge du système. Cela revient à définir des conditions sur les attributs mêmes de l'objet (transmutation automatique : le système prend des "initiatives conditionnées"). Une pré-condition sur les attributs qui caractérisent un objet peut traduire une modification topologique de l'objet, ce qui entraîne alors l'application d'un transmuteur. Dans l'exemple de l'arc, la transmutation intervient s'il possède deux tangences avec deux segments d'un contour et que la matière se trouve du "bon" côté (si la matière se trouve de l'autre côté, c'est un congé).

Un déclenchement explicite provient d'une demande de transmutation à l'initiative de l'opérateur. Le système n'a aucune déduction à faire, il connaît le nouveau type d'objet et par conséquent, le transmuteur à utiliser. Dans ce cas, il sera même possible d'engager le dialogue si des informations viennent à manquer. Avant de transmuter un arc tangent à un seul segment, le dialogue peut s'engager pour obtenir un deuxième segment tangent afin d'autoriser la transmutation (en accord avec la matière). Ces transmutations peuvent être utilisées pour avoir différentes modélisations pour un même objet (un segment peut être un segment mais aussi une courbe sur simple demande). L'opérateur choisit la représentation la mieux adaptée à son travail sans aucune contrainte imposée par le système.

L'importance des transmuteurs lors de la mise en place des objets ne doit pas cacher leur utilité dans la conservation de la validité du modèle. Pour mettre en évidence ce cas, nous nous appuyons sur un problème classique traité en modélisation : le problème du passage de la rainure à une marche en fonction du contexte (cas limites). Il s'agit de satisfaire la géométrie par une modification de la topologie [GOS 88].

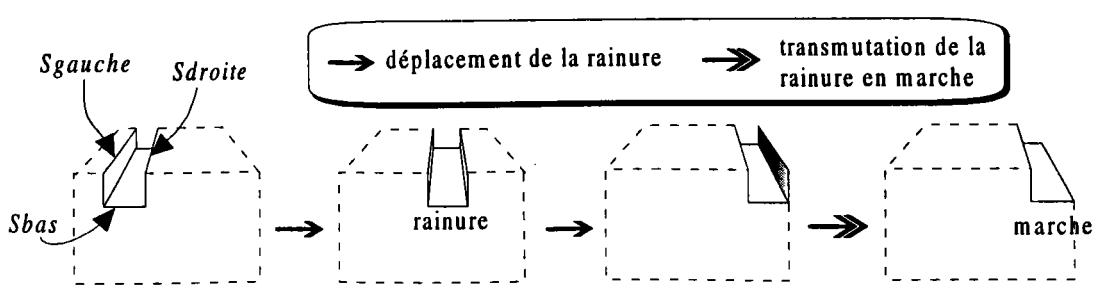


Figure IV.10. *Modification de la topologie par transmutation.*

Dans [WOL 91], une caractéristique est décrite à l'aide des surfaces dites fonctionnelles qui représentent sa "frontière". En nous inspirant de cette définition, nous avons choisi de représenter la rainure débouchante par certaines surfaces fonctionnelles sur lesquelles elle s'appuie : *Sgauche*, *Sbas* et *Sdroite*. Nous avons spécifié la transmutation d'une rainure en marche par la disparition de la surface *Sgauche* ou *Sdroite*. Dans l'exemple, la disparition de la surface *Sdroite* entraîne automatiquement la transmutation de la rainure en une marche définie par les surfaces fonctionnelles : *Sgauche*, *Sbas*.

Grâce aux transmuteurs, l'opérateur peut s'affranchir d'un arbre d'héritage trop souvent restrictif. Il se déplace alors dans un graphe de transmutations qui ne le confine pas dans un mode de pensée. Ici, les transmuteurs sont utilisés par le système pour corriger, pour aller plus loin dans la satisfaction de la validité du modèle. L'évolution des objets se fait d'autant plus en douceur que derrière chaque transmuteur se cache un comportement au niveau du dialogue pour montrer à l'utilisateur la transformation. Ce comportement du dialogue est une conséquence d'une action de l'opérateur (changer l'objet) : c'est un réacteur. Dans la majorité des cas, l'application d'un transmuteur entraîne l'application d'un réacteur. La transmutation d'un objet se fera donc couplée à un réacteur pour informer l'opérateur des déductions et évolutions du modèle.

3.4. Les propriétés des objets.

Après avoir spécifié un objet par une représentation, sa création par des producteurs, son type par des transmuteurs, il faut définir comment l'affiner. Pour cela, nous introduisons les propriétés dont le rôle est de définir des relations. Nous considérons les propriétés comme des entités à part entière. Cette séparation vis-à-vis des objets permet la spécification des propriétés, avant tout, par leur sémantique. Cette sémantique s'appuie sur une description générique indépendante des objets. Les données nécessaires à cette description ne sont pas internes à la propriété car leur construction dépend des objets concernés. Il est alors indispensable de décomposer les relations en autant de données que nécessaires.

Une propriété P est donc définie par :

- un ensemble $\{ D_1, .., D_n \}$ de données;
- une fonction d'évaluation Fe;
- une fonction de création Fc;
- des caractéristiques internes.

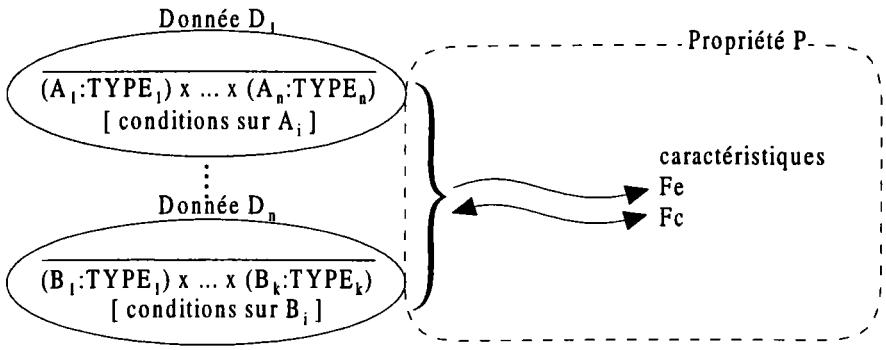


Figure IV.11. *Définition d'une propriété.*

Les données D_i représentent les paramètres nécessaires pour définir la propriété. On trouve, pour chaque propriété, autant de données que l'impose son arité. Une propriété entre n objets nécessite la vérification des n données qui dépendent chacune d'un des objets à mettre en relation.

La fonction d'évaluation Fe représente uniquement le critère que doivent vérifier les données pour que la propriété soit vraie. C'est la description de son domaine de validité.

$$\begin{array}{lll} Fe : D_1 \times .. \times D_n & \longrightarrow & \{ \text{VRAI, FAUX} \} \\ \quad (d_1 , .. , d_n) & \longmapsto & Fe\,(d_1, .., d_n) \end{array}$$

Lorsqu'une propriété est vérifiée, on dit que les données d_i sont complémentaires. Cette fonction est utilisée lorque l'on désire détecter une propriété.

La fonction de création Fc a pour objectif de modifier les données de telle façon que le critère représenté par la fonction Fe soit VRAI.

$$\begin{array}{lll} Fc : D_1 \times .. \times D_n & \longrightarrow & (D_1 \times .. \times D_n)^m \\ \quad (d_1 , .. , d_n) & \longmapsto & Fc\,(d_1, .., d_n) \end{array}$$

Généralement cette fonction ne modifie qu'une seule donnée pour faciliter la compréhension de la modification. Mais ceci est un cas particulier de notre approche qui permet de modifier plusieurs données. Pour donner la ou les solutions, la fonction Fc peut tenir compte des données intiales (sur la figure IV.11, la fonction Fc est illustrée par une double flèche pour mettre en valeur cet aller-retour). Le cas échéant plusieurs solutions peuvent être évaluées (dans la définition de la fonction Fc, la variable *m* indique un espace de solutions à m dimensions), à l'opérateur de faire son choix.

Les caractéristiques sont des attributs destinés à définir pleinement la propriété. On peut citer par exemple le fait qu'une propriété soit symétrique.

Nous nous intéressons plus particulièrement aux relations unaires et binaires. Comme exemple, nous avons choisi de décrire les propriétés "unitaire", "horizontal" et "parallèle".

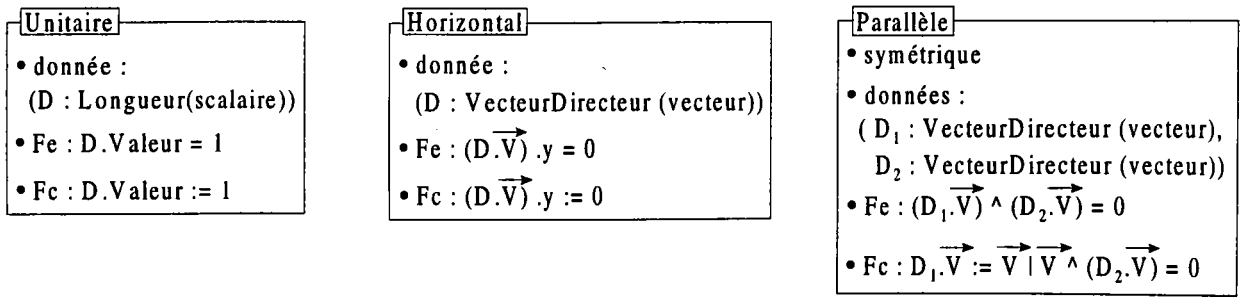


Figure IV.12. *Exemples de propriétés.*

La propriété *Unitaire* est décrite à partir d'une donnée *Longueur* de type *scalaire*. On remarque qu'une donnée possède un nom (*Longueur*) et un type (*scalaire*). Le nom détermine la donnée de manière unique et le type indique ce qu'elle contient. L'identificateur D est l'identificateur de la donnée à l'intérieur de la propriété. Par extension, on parle également de la donnée D. La fonction d'évaluation se contente de vérifier que la longueur est égale à un (*D.Valeur = 1*). La fonction de création fixe la valeur de la longueur à un (*D.Valeur := 1*). Le nom *Longueur* de la donnée est très important, il détermine les objets qui peuvent posséder cette propriété. Tout objet qui fournit la donnée longueur peut posséder la propriété unitaire. Cette propriété est unaire (un objet concerné), ce qui explique qu'il n'y ait qu'une seule donnée nécessaire.

La seconde propriété *Horizontal* est décrite par une donnée *VecteurDirecteur* de type *vecteur* (un vecteur est un couple de coordonnées, on suppose que le vecteur est unitaire). La fonction d'évaluation vérifie que l'ordonnée du vecteur est nulle (D.V est le vecteur et D.V.y est l'ordonnée du vecteur). La fonction de création fixe cette ordonnée à zéro. Là encore, la propriété est unaire.

La troisième propriété est la propriété *Parallèle*. Elle s'appuie sur la même donnée que la propriété *Horizontal* : la donnée *VecteurDirecteur*. Nous avons choisi de la spécifier par la colinéarité de deux vecteurs directeurs unitaires. La propriété est binaire, ce qui impose sa répartition sur deux données : D_1 et D_2. La fonction d'évaluation vérifie que les deux vecteurs sont colinéaires (produit vectoriel égal à zéro). Pour la fonction de création un choix a été effectué, c'est la donnée D_1 qui est modifiée. On aurait pu décrire un dialogue pour que l'opérateur choisisse la donnée à modifier. L'intérêt de cette propriété est qu'elle s'appuie sur la même donnée que la propriété *Horizontal*. C'est une simplification importante, si un objet construit la donnée *VecteurDirecteur*, il supporte automatiquement ces deux dernières propriétés.

Ces exemples montrent clairement l'indépendance des propriétés par rapport aux objets. Seules les données interviennent dans le calcul du critère et aucune mention n'est faite de leur provenance. Elles sont fournies par les réacteurs intrinsèques à partir de la représentation de l'objet. Nous décrivons ces comportements dans le paragraphe suivant.

3.5. Les réacteurs.

Les objets ont des comportements différents selon les actions qui se produisent et selon l'environnement avec les autres objets. Les réacteurs sont chargés de représenter ces comportements. Ils s'appliquent à un objet sans en changer le type.

Par la suite, nous distinguons deux familles de réacteurs :

- les réacteurs orientés données (réacteurs intrinsèques). Ils représentent essentiellement une modification de l'état de l'objet, leur rôle principal est de définir les données des propriétés;
- les réacteurs orientés apparence (réacteurs extrinsèques). Ils définissent plus particulièrement la présentation d'un objet par rapport aux actions extérieures.

3.5.1. Les réacteurs intrinsèques.

Ils sont étroitement liés aux propriétés d'un objet. Ils fournissent les données nécessaires à la détection et à la création des propriétés. Par suite, chaque réacteur intrinsèque est associé à une donnée. On peut considérer que les réacteurs forment des passerelles entre les objets et les propriétés. Le fait d'ajouter aux objets de type α un réacteur intrinsèque associé à une donnée D, entraîne automatiquement la prise en compte de ces objets par les propriétés qui utilisent cette donnée.

Un réacteur intrinsèque R est défini par :

- le domaine Dr;
- la fonction de réaction Fr;
- la fonction de traduction Ft.

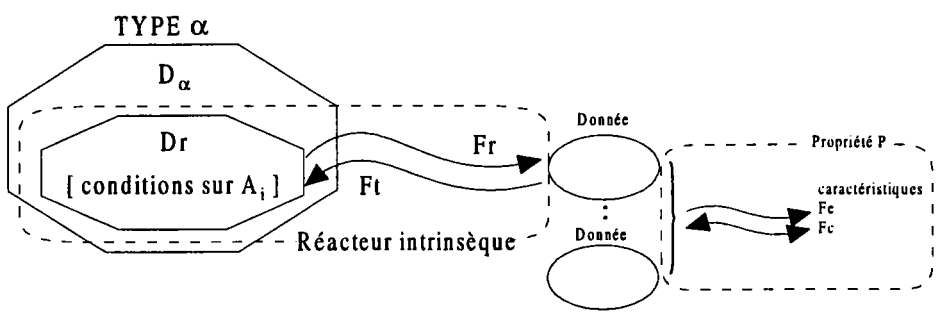


Figure IV.13. *Définition d'un réacteur intrinsèque.*

Le domaine Dr est une restriction du domaine initial des objets ($Dr \subset D_\alpha$). Dans la majorité des cas, le domaine Dr est le domaine initial D_α. Il exprime quand et à quelles conditions un objet doit être pris en compte pour la détection des propriétés.

La fonction de réaction Fr permet de construire la donnée d'une propriété à partir de la représentation de l'objet. Elle prépare l'objet pour la propriété.

Fr :	Dr	$\longrightarrow$	D_i	(avec D_i une donnée d'une propriété)
	o	$\longmapsto$	Fr (o)	

De cette manière, l'objet est automatiquement pris en compte pour le calcul des propriétés qui utilisent la donnée D_i.

La fonction de traduction Ft permet de modifier un objet à partir de la donnée fournie par une propriété. Elle interprète la mise en place de la propriété pour un objet. Les données fixées par la propriété sont traduites dans la représentation de l'objet et ce, en fonction de la représentation courante.

Ft :	$D\alpha$	x	D_i	$\longrightarrow$	$D\alpha$	(avec Di une donnée d'une propriété)
	(o	,	d)	$\longmapsto$	Fc (o)	

De cette manière, l'objet est automatiquement modifié en accord avec les contraintes indiquées par la propriété.

Pour illustrer notre propos, nous intégrons à la définition des objets SEGMENTS et CERCLES des réacteurs intrinsèques afin de gérer les propriétés que l'on a décrites dans le paragraphe précédent (*Unitaire*, *Horizontal* et *Parallèle*). L'objet SEGMENTS est représenté par un couple de points (origine et extrémité) et l'objet CERCLES par un centre et un rayon.

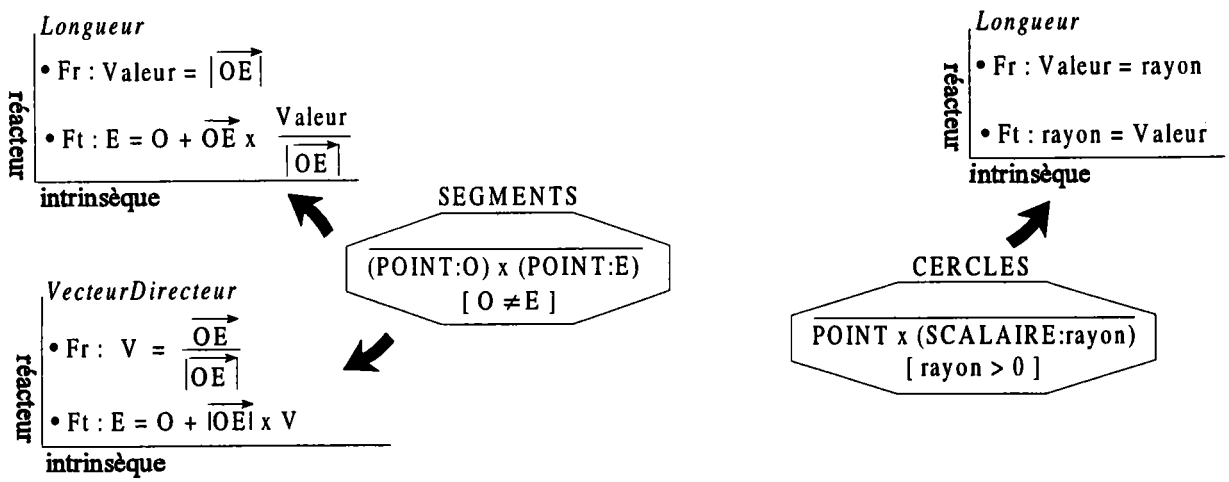


Figure IV.14. *Des réacteurs intrinsèques (le signe ↗ indique les associations comportements-objet).*

Pour gérer la propriété *Unitaire*, il suffit de décrire dans la définition des objets comment obtenir la donnée *Longueur*. Pour cela, nous ajoutons un réacteur intrinsèque du nom de la donnée (c'est un moyen simple pour associer les réacteurs aux données). Ce réacteur contient la fonction de réaction qui calcule *Valeur* l'attribut de la donnée *Longueur*. Pour SEGMENTS, il s'agit de calculer la longueur du segment et pour CERCLES, il s'agit du rayon. Quant à la fonction de traduction, elle interpréte la mise en place de la propriété après la modification de l'attribut *Valeur* de la donnée *Longueur*. Pour SEGMENTS, l'extrémité est recalculée pour que la longueur soit égale à *Valeur*. Remarquons que l'on aurait pu décrire un dialogue pour que l'opérateur puisse choisir le point à modifier de l'origine ou de l'extrémité. Pour CERCLES, on fixe simplement le rayon. Si une autre propriété utilise la même

donnée *Longueur*, aucune modification n'est nécessaire car ces réacteurs ne dépendent pas des propriétés.

Pour gérer les propriétés *Horizontal* et *Parallèle*, il suffit de décrire dans la définition du segment comment obtenir la donnée *VecteurDirecteur*. Nous ajoutons donc un réacteur intrinsèque. Il contient la fonction de réaction, qui calcule le vecteur *V* en fonction de la représentation du segment, et la fonction de traduction, qui va interpréter la mise en place de la propriété. Ici, après modification du vecteur par la propriété, l'extrémité du segment est recalculée pour que le vecteur directeur soit celui qui est fixé par la propriété. Là encore, on aurait pu décrire un dialogue pour que l'opérateur puisse choisir le point à modifier. Ces deux propriétés illustrent comment à l'aide d'un seul réacteur, les objets SEGMENTS se voient enrichis de plusieurs propriétés.

Définir une propriété consiste donc à exprimer sa sémantique et à préciser les traitements que l'on peut effectuer (détection, création ...). Chaque traitement est défini par l'ensemble des réacteurs à associer pour le réaliser. Dans la recherche d'objets vérifiant une propriété, les réacteurs sont utilisés pour filtrer les objets candidats. En effet, pour une propriété binaire P dont les données sont D_1 et D_2, deux objets O et O' sont susceptibles de posséder P si l'un possède un réacteur intrinsèque R_1 l'associant à D_1 et l'autre un récateur intrinsèque R_2 l'associant à D_2 : R_1 et R_2 sont dits complémentaires.

Le travail sur les réacteurs intrinsèques et les propriétés est facilité par le lien étroit qui existe entre les producteurs et les objets. L'avancée des producteurs est immédiatement répercutée sur la représentation de l'objet, ce qui permet une détection "à la volée" des propriétés tout au long de la production (des dialogues). De même, au prix d'une formalisation précise mais difficile à réaliser, le producteur doit pouvoir être influencé par les contraintes fixées sur la représentation de l'objet et ce, à cause d'une (ou plusieurs) propriété(s) demandée(s) par l'opérateur.

3.5.2. Les réacteurs extrinsèques.

Les réacteurs extrinsèques apportent une aide à l'opérateur. Par des modifications de l'apparence propre à l'objet, ils guident l'opérateur dans l'avancée des dialogues. Par des ajouts d'informations sur les objets, ils informent l'opérateur des interprétations du système. Leur rôle est de traduire à l'opérateur des traitements effectués par le système sur les objets.

Un réacteur extrinsèque R est défini par :

- le domaine Dr. Il représente le domaine sur lequel s'applique le réacteur;
- la fonction de réaction Fr; elle construit l'apparence à partir de la représentation de l'entité à laquelle le réacteur est associé.

Fr	: Dr	⟶	Modèle	(par exemple, un modèle d'affichage)
	o	⟶	Fr (o)	

Les réacteurs extrinsèques prennent toute leur importance lors des interactions floues. Par exemple, il s'agit de définir la réaction d'un producteur à une réponse (comportements liés aux producteurs), d'indiquer comment l'opérateur doit répondre (comportements liés au dialogue), ... Les réacteurs extrinsèques sont chargés de représenter tous ces comportements. Les déclenchements (et donc les utilisations) de ces réacteurs sont donc nombreux et variés :

- changement d'état de l'objet. La modification des attributs de représentation de l'objet doit entraîner une mise à jour de son apparence. Cette mise à jour est à la charge des réacteurs extrinsèques associés à l'objet;
- exécution d'un producteur. Lors de l'utilisation d'un producteur, un réacteur extrinsèque indique l'avancée des dialogues à l'utilisateur en fonction de l'état du producteur. Il s'agit d'aider l'utilisateur à continuer ses dialogues en indiquant ce qui est déjà défini et ce qui reste à définir pour réaliser complètement l'objet avec ce producteur. Le fait que le producteur maintienne la représentation de l'objet à jour, garantit un dialogue dynamique (l'objet apparaît au fur et à mesure de sa création) mais cela ne suffit pas pour poursuivre la création. Les réacteurs extrinsèques associés aux producteurs indiquent les interactions à effectuer pour terminer l'objet (ces aides peuvent être textuelles, graphiques, ...);
- application d'un transmuteur. Les réacteurs extrinsèques sont utilisés pour informer l'opérateur d'une transmutation et pour l'aider à la comprendre. Ils sont très importants lors de la création des objets qui entraîne parfois une transmutation immédiate. Par exemple une ellipse peut se transformer en cercle immédiatement après sa création et l'utilisateur est ainsi informé de cette transmutation qui peut être très importante à ses yeux (la pièce est de forme circulaire, ...);
- création d'une propriété. Les réacteurs sont utilisés pour indiquer les interprétations du système. Un réacteur extrinsèque associé à une propriété ou à un réacteur intrinsèque informe l'opérateur de la détection ou de la création d'une propriété par un affichage adéquat;
- demande explicite de réacteur. L'opérateur choisit le réacteur à appliquer et par suite, l'apparence de l'objet ou les aides qui lui seront apportées.

Dans l'exemple du cercle, nous associons des réacteurs extrinsèques au type *CERCLES* et au producteur *CercleCentrePtDePassage*. Le but est de définir l'affichage pendant la création du cercle :

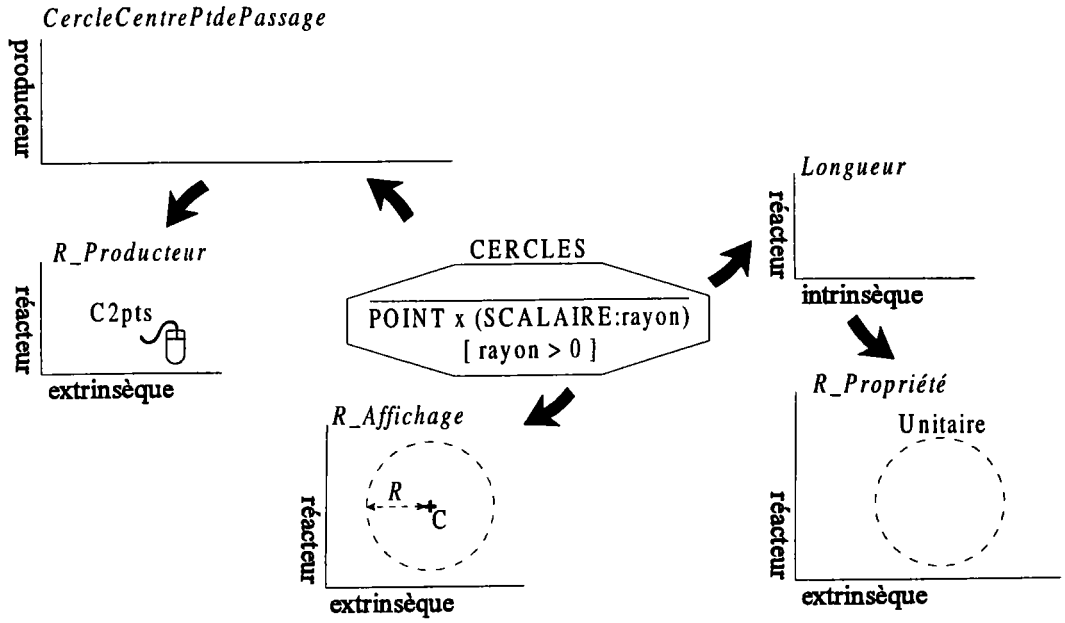


Figure IV.15. *Ajout de réacteurs extrinsèques à l'objet CERCLES (le signe ➚ indique les associations comportements-objets et comportements-comportements).*

Trois réacteurs ont donc été ajoutés :

- *R_Producteur* qui est associé au producteur *CercleCentrePtDePassage*; il définit une aide visuelle très simple indiquant à l'opérateur le producteur utilisé par un message lié au pointeur de désignation. Des réacteurs plus spécifiques auraient pu être décrits comme dans le cas de la fonction de production, le réacteur dépendant alors de l'état du producteur (l'aide varie selon les dialogues engagés);
- *R_Affichage* qui est associé directement à l'objet CERCLES; il décrit comment un cercle doit être visualisé. Comme les producteurs sont chargés de maintenir la représentation à jour tout au long des dialogues, l'utilisateur visualise grâce à ce réacteur, l'avancée de la construction. Nous avons choisi une seule présentation, on aurait pu en définir d'autres mais une seule peut être utilisée pour un affichage donné (comme nous le verrons, l'utilisateur peut choisir le réacteur à utiliser ou ce choix peut être fait en fonction des aides que l'on désire apporter);
- *R_Propriété* qui est associé au réacteur intrinsèque *Longueur* (ce réacteur construit l'attribut Valeur nécessaire à la propriété *Unitaire* qui vérifie que Valeur est égal à un). Il indique comment le système doit informer l'opérateur de la détection de la propriété *Unitaire* sur le cercle concerné. Ce réacteur est associé au réacteur intrinsèque car il est très dépendant de l'objet concerné. En effet, la propriété ne connaît pas l'objet qui a fourni la donnée et ne peut donc afficher d'aide dépendant de cet objet (on peut définir des réacteurs extrinsèques sur une propriété mais les informations ne peuvent tenir compte des objets mis en relation). C'est pourquoi, ici, l'affichage du mot 'Unitaire' sur le cercle doit être réalisé à partir du réacteur intrinsèque.

Il faut toutefois noter que, si l'indépendance des propriétés par rapport aux objets est très satisfaisante, elle pose un problème lorsque l'on considère l'indépendance des objets par rapport aux propriétés. Pour

une donnée d'un objet, il est a priori impossible de savoir quelle propriété est détectée puisque par définition une donnée est utilisable par toutes les propriétés concernées. Par conséquent dans le cas de l'objet SEGMENTS qui dispose d'une donnée (*VecteurDirecteur*) utilisée par deux propriétés, il faut imaginer un mécanisme pour associer des réacteurs extrinsèques dédiés à une propriété. Dans le cas contraire, on ne pourrait pas informer l'opérateur de la détection d'un segment horizontal ou d'un segment parallèle puisque le réacteur intrinsèque ne connaît pas la propriété.

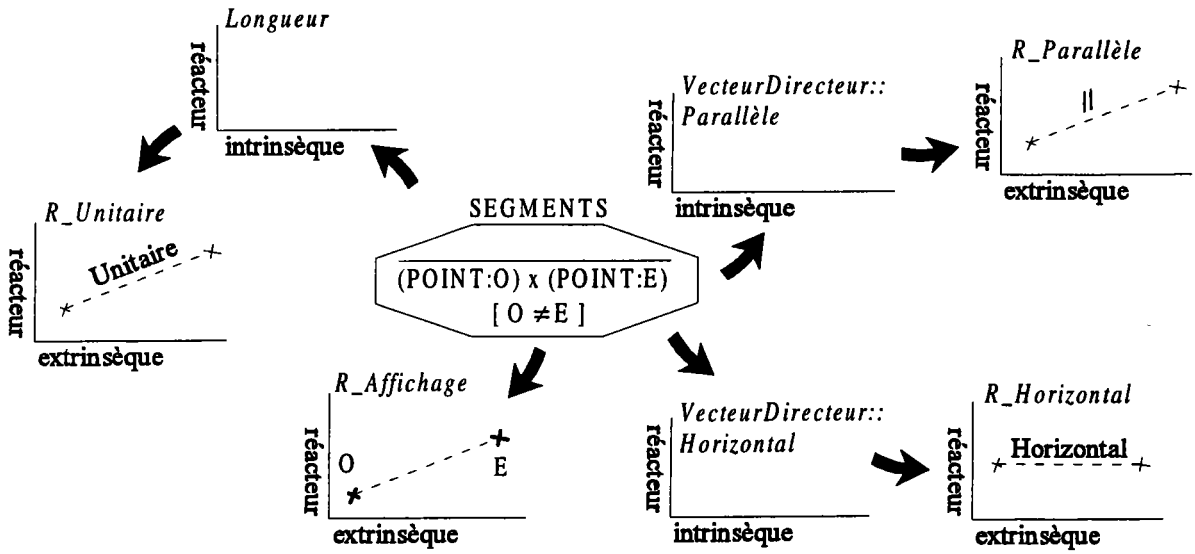


Figure IV.16. *Ajout de réacteurs extrinsèques à l'objet SEGMENTS (le signe ↗ indique les associations comportements-objets et comportements-comportements).*

Nous avons choisi de préciser les réacteurs intrinsèques en fonction de la propriété : **Donnée::Propriété**. Ainsi, le réacteur *VecteurDirecteur::Parallèle* est dédié à la propriété *Parallèle* alors que le réacteur *VecteurDirecteur::Horizontal* est associé à la propriété *Horizontal*. Nous insistons sur le fait que le réacteur intrinsèque original *VecteurDirecteur* est conservé sans aucune modification, les deux réacteurs précédents ne sont que des entités, des passerelles permettant de le préciser pour certaines propriétés. De cette manière, en associant le réacteur extrinsèque *R_Parallèle* au réacteur intrinsèque *VecteurDirecteur::Parallèle*, on indique comment le système doit informer l'opérateur de la détection de la propriété *Parallèle* sur le segment concerné (afficher le signe // au milieu du segment). De même pour la propriété *Horizontal*. Pour la donnée *Longueur*, aucune précision n'est apportée car une seule propriété l'utilise dans notre exemple. Un outil de conception est envisagé pour faciliter la création de ces réacteurs car il peut prendre en charge toutes ces précisions et peut même aider à la mise en place des réacteurs extrinsèques en indiquant de lui-même les données partagées.

4. LA GÉNÉRATION DU DIALOGUE.

Nous allons mettre en évidence la dualité qui existe dans la définition du dialogue. À l'aide du langage graphique, défini dans le chapitre précédent, le système propose l'ensemble des dialogues déduits du modèle et l'opérateur peut adapter le résultat à sa propre vision et à son domaine de connaissance. Nous montrons ainsi qu'il est possible de déduire en grande partie le dialogue en interprétant le modèle de C.F.A.O. représenté par les comportements.

L'étape de génération est très importante car elle représente le passage entre la définition du modèle et son utilisation. À partir du modèle, le système est capable de générer un certain nombre de dialogues et d'aides au dialogue :

- les actions de création en s'appuyant sur la définition des producteurs;
- les actions de détection et de création des propriétés en utilisant à la fois les producteurs et la définition des propriétés par les réacteurs intrinsèques;
- les aides contextuelles tout au long des différentes interactions. Elles sont fournies par les réacteurs extrinsèques.

Cette déduction permet d'obtenir un premier prototype de l'application désirée sans autre intervention. Bien entendu, ce prototype ne peut tenir compte des spécificités de l'application mais utilise les rôles de base de chacun des objets. Il reste alors à l'opérateur à préciser chacun des paramètres de ces dialogues, voire d'en créer de nouveaux, grâce au langage graphique (les actions qui ne peuvent être déduites du modèle restent de toutes façons à la charge du programmeur d'interfaces : zoom, ...). Ce langage permet de préciser le prototype obtenu pour finalement aboutir à l'application désirée. Ces précisions sont apportées par l'opérateur alors que le système est déjà opérationnel, ce qui permet le test immédiat et une évaluation immédiate de ces ajouts.

Pour illustrer les possibilités de génération offertes par cette approche, nous montrons l'intérêt de chacun des comportements en mettant en avant les possibilités de déduction que l'on envisage. Tout au long de cette présentation, nous utilisons des exemples en filigranes pour mettre en évidence la génération et son intérêt.

4.1. Les producteurs.

C'est un comportement qui contient comme données un domaine de création (attributs et conditions) et la fonction de création (il peut également intégrer des liens avec des réacteurs). Le producteur ainsi défini, il faut déterminer l'action globale qui lui est associée. Il s'agit, à partir du domaine de création et de bibliothèques d'actions, de construire les séquences d'interactions (et leurs contraintes) pour aboutir à la fonction de création. Ces interactions sont obtenues à l'aide d'une analyse des conditions spécifiées sur les attributs.

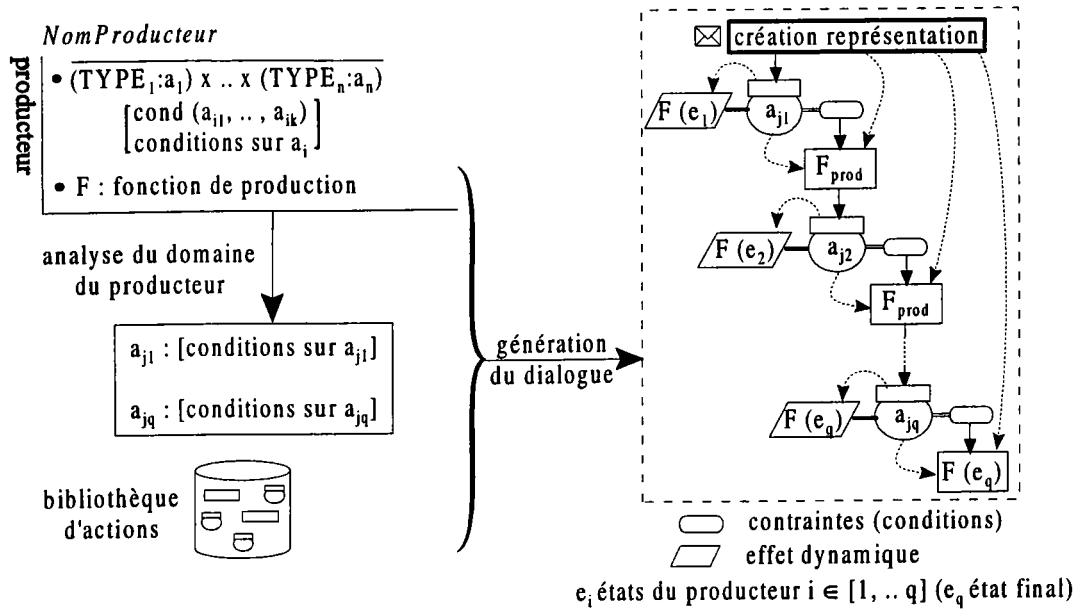


Figure IV.17. Génération du dialogue de création à partir d'un producteur.

La génération doit garantir au mieux la répartition des contraintes et les appels à la fonction de production. Les conditions des attributs (cond (a_{j1}, .., a_{jk})) sont analysées (par réécriture par exemple) pour ordonner les interactions qui correspondent aux attributs en fonction des dépendances qui existent sur les conditions. Les conditions sont distribuées sur chacun des attributs et un ordre est ainsi déterminé pour les interactions correspondantes (a_{j1}, .. , a_{jq}). Au début de l'action globale, l'objet est créé grâce à une action de création de sa représentation; le résultat de cette action est le résultat de l'action globale (au début sa représentation est vide mais elle va évoluer au fur et à mesure des interactions et, à la fin, elle contient l'objet construit). À chaque étape où un appel à la fonction de production est possible (état du producteur correspondant à un état cohérent de la représentation de l'objet), une action correspondante F (état) est créée et insérée dans l'action globale. La fin de l'action globale est marquée par la création finale de l'objet par l'action F (état final). De même, un effet dynamique correspondant à la fonction de production est ajouté aux interactions. De cette manière, l'opérateur peut suivre grâce à la mise à jour de la représentation l'évolution de l'objet pendant l'exécution d'une interaction (il voit l'objet tel qu'il sera lors de la validation de l'interaction).

Dans la figure suivante, nous considérons le producteur *Cercle3Pts*. La première étape consiste à décomposer les pré-conditions sur les attributs en contraintes portant sur les interactions : on peut faire appel à un mécanisme de réécriture qui s'appuie sur la dépendance des attributs. Ainsi, aucune contrainte ne porte sur la première interaction (attribut pt_1 choisi en premier). Cette phase de réécriture impose déjà un début d'ordonnancement pour le dialogue. La seconde étape consiste à déduire le séquencement réel du dialogue. La première action générée est la création de l'allure générale de l'objet : il s'agit de créer un cercle selon la représentation, ici le couple (centre : point quelconque, rayon : réel positif). Les contraintes déduites lors de la phase de réécriture couplées à une bibliothèque d'actions permettent d'ordonner les interactions nécessaires à la création du cercle et de calculer exactement la représentation du cercle.

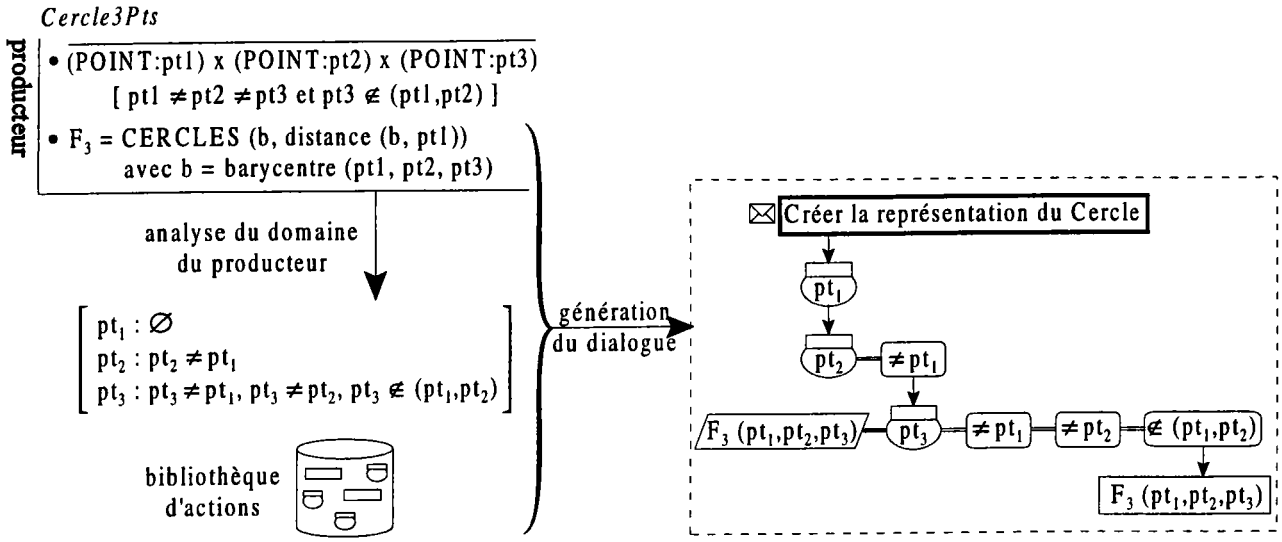


Figure IV.18. *Génération du dialogue de création d'un cercle à partir de trois points.*

Un effet est ajouté à l'interaction pt_3 pour mettre à jour la représentation. La fonction de production, F_3, ne peut être appelée que lorsque les trois points sont connus, ce qui représente l'état final du producteur. Tout au long de l'interaction, à chaque point proposé par l'opérateur, la représentation est modifiée pour prendre en compte le point candidat. L'opérateur peut alors se rendre compte du résultat. Il est à noter que les contraintes qui portent sur ce troisième point sont redondantes. Le fait de vérifier qu'il n'appartient pas à la droite (pt_1, pt_2) implique qu'il est différent de pt_1 et de pt_2. Ce fait montre bien l'importance de l'analyse du domaine du producteur (pour éliminer ou déterminer des contraintes) et des difficultés sous-jacentes.

Un producteur représente une façon de créer un objet auquel il est attaché. Ainsi, en accord avec notre modèle d'interaction, toute action globale issue ou générée à partir d'un producteur d'un objet de type α peut être considérée comme un local à toute interaction demandant un objet de type α. On enrichit la bibliothèque des interactions au fur et à mesure de la génération du dialogue et par suite, les actions globales déjà définies dans le système (lors de la génération du producteur *Cercle3Pts* d'un cercle, un lien local vers le menu correspondant est ajouté à l'interaction de type *CERCLES* de la bibliothèque d'interactions).

Dans le même ordre d'idée, l'analyse des producteurs permet de poursuivre un dialogue qui pourrait paraître mal engagé dans la mesure où la réponse retournée par l'opérateur ne correspond pas à la réponse attendue. Si l'objet retourné par l'opérateur est un des attributs d'un producteur de l'objet attendu, le système oriente le dialogue vers ce producteur : le dialogue est alors initié par les producteurs. Reprenons l'exemple du cercle : le premier objet autorisé pour la construction d'un cercle par le producteur *CercleCentreRayon* est un point ou un scalaire. Lors d'une interaction demandant un cercle à l'opérateur, le système déclenche la création d'un cercle de manière implicite au cas où l'opérateur fournit un point ou un scalaire. Dans le cas contraire, la création est impossible et l'objet doit être rejeté. L'utilisation d'un producteur représente ici un manque d'informations pour répondre à

l'interaction : ce comportement est donc local à l'interaction. En évitant le recours à un menu, le producteur facilite l'accès au concept d'effet local.

En fait, chaque objet peut contenir plusieurs producteurs. Il peut paraître important de disposer d'un producteur usuel pour chaque objet. Fixer ce producteur de base ne répondra pas toujours à l'attente de l'opérateur. Ainsi, il s'avère intéressant d'observer la fréquence d'utilisation des producteurs, dans un souci de fidélité envers l'opérateur pour garantir une anticipation cohérente lors d'interactions floues. Une autre solution est d'effectuer une composition de producteurs. Nous présentons cette possibilité dans le paragraphe suivant.

4.2. La composition de producteurs.

Pendant la phase de dialogue de création, les producteurs doivent encore intervenir pour choisir le mode de création en fonction de l'avancée des dialogues. Le grand nombre de producteurs associé à un objet est un avantage dans notre approche. En effet, la construction d'un objet peut se faire non seulement par un producteur donné mais également à partir d'un ensemble de producteurs. L'objet est créé indépendamment du producteur, puis affiné par le choix implicite du producteur désiré.

Dans l'exemple du cercle, l'opérateur peut créer un cercle sans choix préalable du producteur. Le choix est effectué par le système en fonction des données désignées par l'opérateur.

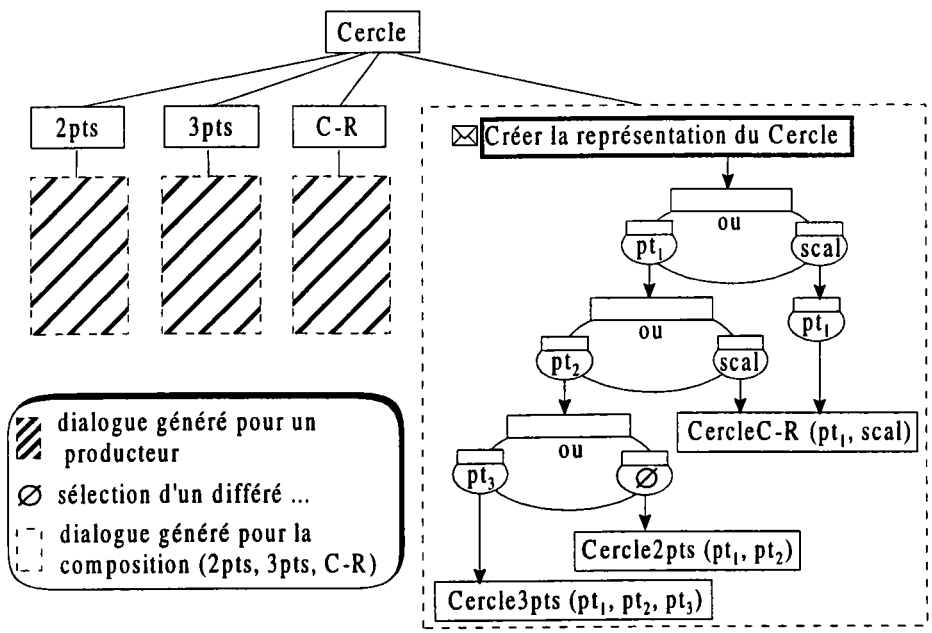


Figure IV.19. *Dialogue généré à partir d'une composition de producteurs (les contraintes ne sont pas représentées pour ne pas surcharger la figure).*

Par exemple, si le premier objet désigné est un point, les trois producteurs restent candidats alors que si c'est un un scalaire, seul le producteur *CercleCentreRayon* reste candidat.

La composition est une approche originale dans laquelle l'opérateur n'a pas à choisir au préalable un dialogue, ce choix se fait au fur et à mesure de sa réflexion et non avant, le système ne proposant que

les producteurs encore possible pour un état du dialogue donné. L'opérateur se trouve aidé dans sa construction par les possibilités qui lui restent. La génération du dialogue reprend les mêmes étapes que pour un producteur simple.

Des cas d'ambiguïtés peuvent toutefois apparaître et notamment des attributs semblables ou des fonctions de création différentes pour un même état du dialogue. Dans ce cas, il sera toujours possible de choisir le producteur à utiliser grâce à des styles d'interactions standard. Pour un état du dialogue, l'utilisateur peut donc évaluer les différentes représentations qu'il peut obtenir en changeant de producteur.

4.3. Les transmuteurs.

Comme nous l'avons vu, une transmutation est appliquée à l'initiative de l'utilisateur (explicite) ou du système (implicite).

Pour une transmutation explicite, la description contenue dans le transmuteur donne les informations nécessaires pour générer l'action globale destinée à effectuer la transmutation. Le menu correspondant peut être de type général, interaction verbe-objets. L'action globale doit alors contenir l'interaction de sélection de l'objet pour lui appliquer la transmutation. Les compatibilités de type local sont aisément générées en déclarant local tout menu de construction des objets concernés par le transmuteur. Par contre, le menu peut être de type contextuel, interaction objets-verbe. Dans ce cas, l'action globale se résume à la transmutation proprement dite sans dialogue particulier. L'action globale de transmutation est la même, la seule différence est que dans le premier cas, c'est l'opérateur qui doit fournir l'objet alors que dans le second, c'est le système qui le fournit selon le contexte (objet auquel appartient le menu).

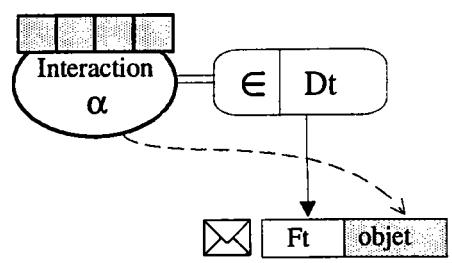


Figure IV.20. *Une action globale de transmutation.*

Toutefois, la génération peut aller beaucoup plus loin avec la déduction de dialogues pour amener une transmutation. Il s'agit de générer une action globale pour qu'un objet O de type α ($O \in D\alpha$) se transmute en un objet O' tel que $O' \in Dt$ avec Dt le domaine de transmutation du transmuteur désiré. L'objectif est de dépasser la simple vérification des conditions de la transmutation. À partir du but que s'est fixé l'opérateur (par exemple la construction du congé à partir d'un arc comme dans la figure IV.9), le système le guide en lui indiquant les dialogues nécessaires, les outils à sa disposition et les conditions à vérifier (par exemple pour le congé, il faut deux tangences et donc modifier l'arc ou d'autres objets). Cette génération est très complexe, même dans des cas simples, mais elle présente un

intérêt non négligeable car elle entre dans la catégorie des aides fournies à l'opérateur, notamment lorsqu'il découvre le système.

Dans le cas de la transmutation implicite, son déclenchement peut être à la charge du modèle ou du dialogue. Pour le déclenchement interne au modèle, le dialogue n'est pas concerné, le simple fait de modifier la représentation déclenche automatiquement les transmuteurs. Par contre, si le déclenchement est à la charge du dialogue, il faut ajouter à toutes les actions de mise à jour de la représentation (fonctions de création du producteur), une action de déclenchement des transmuteurs. La mise à jour entraîne une modification de la représentation, il faut vérifier pour chaque transmuteur si elle appartient au domaine de transmutation (Dt). Dans l'affirmative, la transmutation peut être déclenchée en s'appuyant sur la représentation mise à jour.

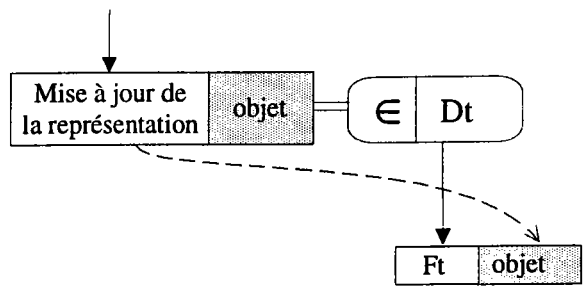


Figure IV.21. *Le contrôle externe de la transmutation.*

Par analogie au contrôle du dialogue que l'on qualifie d'interne ou d'externe, on peut distinguer un déclenchement interne (modèle) ou externe (dialogue) pour les transmuteurs. Lorsque le déclenchement est externe, le concepteur d'interfaces possède une grande latitude quant à la personnalisation des traitements. Cela donne une grande importance au dialogue dans la conservation de la cohérence du modèle. Bien entendu, ces deux types de déclenchements apparaissent complémentaires. Le fait de proposer ces deux types de déclenchement n'oblige pas à choisir obligatoirement un des deux. Il est plus judicieux de les utiliser de façon conjointe : le déclenchement externe convient très bien lors de la création pour contrôler dans le dialogue les transmutations alors que le déclenchement interne est très utile pour automatiser les transmutations à la suite de modifications effectuées par programme sur les objets (un objet se transmute de manière transparente sans intervention explicite).

4.4. Les propriétés des objets.

Les propriétés sont caractérisées par deux phases très différentes : la détection et la création. La génération diffère également selon l'opération que l'on désire réaliser.

La détection est effectuée après une modification de la représentation de l'objet. Il suffit d'évaluer la fonction d'évaluation Fe de la propriété. Tout comme pour les transmuteurs, la responsabilité du déclenchement de la détection des propriétés peut être à la charge du dialogue ou du modèle. Ici, un contrôle externe (par le dialogue) prend tout son sens. Lors d'une interaction floue pour laquelle l'objectif n'est pas clairement défini (en particulier lors de la création de l'objet), de nombreuses

propriétés sont détectées, mais pas toujours souhaitées. Le dialogue qui a le contrôle sur les propriétés peut, par la mise en place d'une ou plusieurs interactions standard, autoriser ou interdire des propriétés, filtrer les objets susceptibles d'être mis en relation, ... Nous préconisons donc d'utiliser, dans les phases de création, un contrôle externe de la détection des propriétés. Pour un segment, on pourra par exemple interdire la détection de l'horizontalité ou au contraire la privilégier. Ainsi, après avoir mis à jour la représentation d'un objet, le dialogue insère une action de détection des propriétés désirées. Pour une propriété donnée, il suffit de vérifier que la représentation de l'objet est dans un état cohérent (représentation suffisament avancée) pour que le réacteur intrinsèque puisse calculer la donnée grâce à la fonction Fr. Il reste alors à rechercher les objets qui possèdent les données complémentaires (réacteurs associés) et à évaluer le critère de détection.

Remarquons que la détection des propriétés se fait au même niveau que les transmutations et peut avoir une influence sur ces dernières (par exemple dans le cas de l'arc qui se transmute en congé, il faut avoir détecté les deux tangences). Il convient donc de détecter les propriétés avant le déclenchement des transmutations. Une transmutation entraîne alors la modification automatique des propriétés qui sont ré-évaluées. Nous travaillons également sur des règles de transmutation des propriétés : lorsqu'un objet transmute, ses propriétés transmutent également selon certains mécanismes.

La création d'une propriété passe par la détermination des objets à mettre en relation. Pour une propriété P qui s'appuie sur des données D_1, .., D_n, il suffit de déterminer pour chaque donnée D_i l'ensemble E_i des types d'objets capables de fournir la donnée D_i par l'intermédiaire d'un réacteur intrinsèque. Ensuite, pour chaque donnée D_i, il faut générer une interaction demandant un objet dont le type appartient à l'ensemble E_i. Enfin, la mise en place de la propriété est effectuée grâce à l'appel de la fonction de création correspondante puis à la traduction du résultat par le ou les réacteurs intrinsèques concernés par la modification.

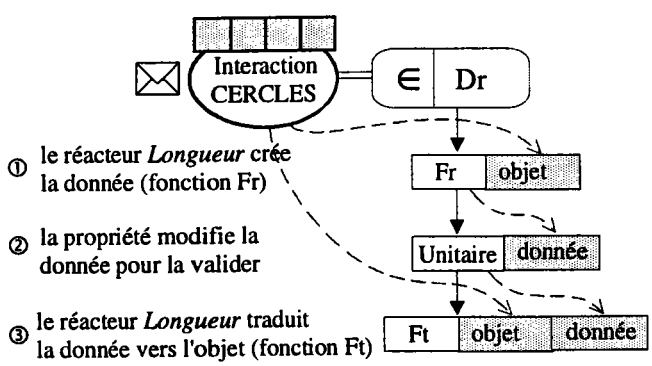


Figure IV.22. *Le dialogue de création de la propriété Unitaire.*

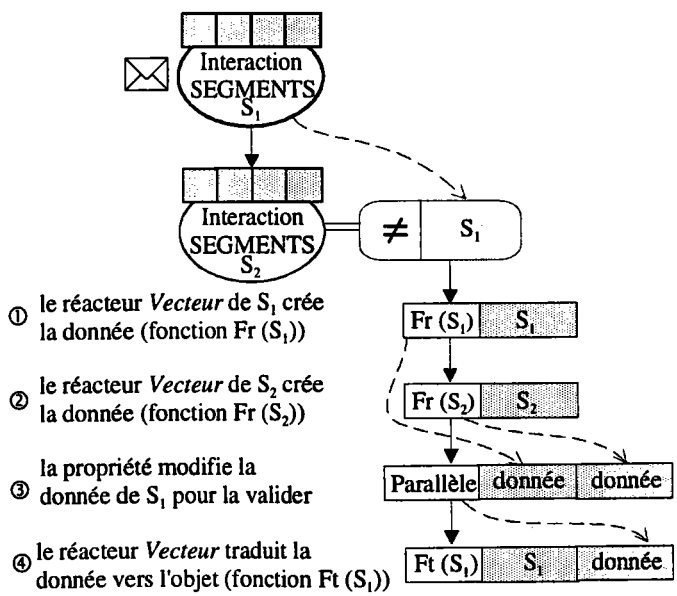


Figure IV.23. *Le dialogue de création de la propriété Parallèle.*

Comme nous l'avons vu dans le paragraphe des réacteurs intrinsèques, il est possible que la traduction de la mise en place de la propriété vers la représentation de l'objet soit elle-même composée de dialogue. La création d'une propriété peut donner plusieurs solutions et les dialogues décrits au niveau du réacteur intrinsèque permettent à l'opérateur de choisir sa solution. Dans ce cas, il suffit d'insérer les dialogues décrits dans le réacteur intrinsèque à la fin de l'action globale de création de la propriété et avant la fonction de traduction. Par exemple, pour un segment dont l'origine est fixe et que l'on désire tangent à un cercle, l'opérateur doit choisir une solution parmi les deux points fournis comme solution pour l'extrémité.

La détection des propriétés lors de la création des objets ou la création d'une propriété sur des objets existants sont des opérations que l'on peut générer de manière quasi systématique. Par contre, il est une autre possibilité plus délicate à réaliser : la création d'un objet qui doit satisfaire un certain nombre de propriétés. On peut se contenter de rejeter les objets qui ne satisfont pas les propriétés mais cette manière de faire va à l'encontre de l'ergonomie et des aides fournies à l'opérateur. Notre but est bien entendu de fournir des informations à l'opérateur pour l'aider à construire son objet avec les propriétés demandées. Par exemple, la création d'un segment que l'on veut horizontal entraîne l'ajout de contraintes sur la deuxième interaction. Par contre, la création d'un cercle unitaire fixe un attribut (le rayon) et supprime donc une interaction dans le dialogue de création. En fait, cela revient à contraindre un producteur par rapport aux propriétés désirées. De nombreuses solutions sont à envisager dans ce domaine qui reste un problème ouvert (construction de dialogues "à la volée", ajout dynamique de contraintes, ...).

Le découpage du modèle en comportements illustre bien les dialogues à plusieurs niveaux qui sont mis en jeu lors de la création des objets. Chaque comportement spécifie les dialogues qui le concernent et l'outil de génération détermine leur séquencement.

4.5. Les réacteurs extrinsèques.

Ces réacteurs sont dans la majorité des cas déclenchés ou paramétrés par le dialogue en fonction de son état. Là encore, la modularité du modèle autorise des générations très différentes. Au delà de la génération simple qui est le déclenchement d'un réacteur extrinsèque à chaque mise à jour d'un objet, nous envisageons des aides de plus haut niveau :

- choix des réacteurs extrinsèques;
- mise en valeur des objets.

Lorsque plusieurs réacteurs extrinsèques sont associés à une même entité (producteur, objet, propriété), la génération consiste en plus à déduire des menus autorisant le choix du réacteur à utiliser. On peut envisager plusieurs réacteurs d'affichage pour un même objet, les menus générés permettent alors de choisir le type de visualisation désiré par l'opérateur. Par exemple, pour un objet, on peut choisir un niveau de détails différent et pour une propriété ou un réacteur intrinsèque, le changement du réacteur permet de choisir la manière avec laquelle la détection de la propriété sera indiquée.

Au cours de la création d'un objet et en utilisant la définition des propriétés, nous pouvons déclencher les réacteurs extrinsèques de mise en évidence des objets complémentaires. À partir de la propriété de vissage décrite entre le type VIS et le type TROU, il est possible, pendant la création d'une vis, de signaler à l'opérateur les trous qui possèdent des données susceptibles de vérifier le critère (diamètre, filetage, ... compatibles). Cette génération est très importante pour situer le travail que fait l'opérateur dans le monde virtuel qu'il construit. Les filtres que nous envisageons pour les propriétés sont également applicables pour ces aides. On peut ainsi privilégier des aides selon certaines propriétés ou certains types d'objets (poser un verre sur une table et pas ailleurs, ...).

Concernant les réacteurs extrinsèques liés aux aides, un travail important de spécification reste à réaliser. Ils ne peuvent se limiter à de simples affichages mais doivent être de véritables documentations en ligne qui tiennent compte du contexte. En particulier, lors d'une composition, ces aides doivent être cumulées pour obtenir des messages cohérents ce qui implique une formalisation très précise.

5. Implémentation et conclusion.

De nombreux obstacles persistent dans le processus de développement des interfaces utilisateurs. Pour obtenir l'interface utilisateur désirée, il est nécessaire de réaliser une programmation (graphique, textuelle, ...) massive. En nous appuyant sur les travaux réalisés dans les deux chapitres précédents (modèle d'interaction et modèles de dialogue), nous avons développé un outil permettant au programmeur d'interfaces de créer un dialogue adapté à un opérateur et de créer l'architecture du logiciel de C.F.A.O. Le seul inconvénient notable de cette approche est que, dans la mesure où l'univers virtuel manipulé par le système est extrêmement complexe, il est pratiquement impossible au programmeur d'interfaces, sans se plonger dans les détails du système (ce que l'on souhaite éviter), d'être assuré de n'avoir oublié aucune possibilité. Notre proposition est de mettre en place un outil profitant au mieux des connaissances du modèle et de l'expérience du programmeur d'interfaces. L'application de ces concepts à la C.F.A.O. est originale étant donné l'importance du modèle des objets et de la complexité de tels systèmes (complexité des traitements, des manipulations, ...).

Notre étude s'inscrit dans le courant des approches orientées modèles (Model-Based Approach) dans lesquelles on déduit d'un ou de plusieurs modèles des informations utiles à un dialogue. Le principal objectif de ces systèmes est de cacher la complexité du développement du dialogue en automatisant toutes les décisions de conception effectuées par le concepteur. Cependant, le faible contrôle qu'ils fournissent sur ces décisions entraînent qu'il est difficile de générer des interfaces de bonnes qualités [SZE 93]. Parmi ces systèmes, on peut citer notamment UIDE qui fournit des mécanismes pour informer l'interface des modifications effectuées dans l'application (essentiellement par des pre et post-conditions appliquées aux fonctionnalités) [GIE 92; FOL 93]. Un autre système, Nephew, s'intéresse plus particulièrement à une infrastructure et des protocoles à travers lesquels l'interface peut interroger l'application (formalisation du composant *Interface Avec Application* du modèle d'architecture de Seeheim) [SZE 89].

Le dialogue du système SACADO étant opérationnel à la suite des travaux présentés dans le chapitre 3, nous nous sommes attachés à présenter les différents composants qui permettent de déduire un dialogue à partir des connaissances du modèle. L'intégration, en vraie grandeur, de la déduction du dialogue à partir du modèle nécessite le développement d'un modèle incluant l'ensemble des contraintes. Nous avons choisi de décrire le modèle par un couple données/comportements qui s'apparente au couple données/méthodes des langages orientés objets. On peut trouver un parallèle lointain avec le système GARNET [MYE 90c; MYE 92d] dans lequel les interactions que l'opérateur peut effectuer avec un objet sont encapsulées dans des "interacteurs". De cette manière, en associant un "interacteur" à un objet, le concepteur lui donne une réaction face à l'opérateur. Notre modèle fonctionne également par association de comportements à un objet, comportements qui décrivent sa création (producteurs), sa tranformation (transmuteurs) et ses réactions (réacteurs).

À partir de ce modèle, nous avons montré qu'il était possible de déduire un grand nombre de dialogues et de les présenter sous le même formalisme graphique utilisé pour construire une interface homme-machine (chapitre 3). Le programmeur d'interfaces peut adapter la génération à sa propre vision et à son domaine de connaissance. On peut ainsi considérer que la déduction des actions à partir du modèle permet d'être exhaustif (on n'oublie aucune possibilité du modèle dans le dialogue). Cette description, lorsqu'elle est trop complexe, peut servir de base de travail pour le programmeur d'interfaces (voire l'opérateur) pour aboutir à une interface conviviale, en s'appuyant sur le même formalisme. La présentation dans un formalisme compréhensible par un opérateur non-informaticien permet à celui-ci de l'adapter, tout en étant assuré que toutes les possibilités inhérentes au modèle sont présentes. Cette approche duale dans la description du dialogue, génération par le système et intervention de l'opérateur, répond en grande partie à l'inconvénient de l'automatisation et du faible contrôle sur la génération.

Nous avons proposé des solutions de génération mais tout l'intérêt de notre approche repose sur sa modularité : une nouvelle génération qui s'appuie sur les comportements existants ne remet pas en cause le modèle. Toutes les implantations se voient enrichies de ces nouvelles actions générées. Cela représente une simplification du travail dans bien des cas. Le programmeur de modèles décrit ce qui est ou doit être alors que la génération se charge de déduire comment manipuler ces faits, manipulation que le programmeur d'interfaces peut adapter ou préciser selon les besoins.

Ce chapitre a été l'occasion de présenter une solution au développement du dialogue. Cette présentation avait pour but de rechercher ce que l'on pouvait espérer comme modélisation et comme génération. Une première implémentation dont la description se trouve dans l'annexe D a été réalisée et les résultats obtenus sont prometteurs [GAR 96]. Elle a permis de valider la décomposition du modèle en comportements et d'intégrer certaines générations proposées précédemment. Mais le travail qui reste à accomplir est vaste. Il est essentiel de rechercher une formalisation mathématique précise du modèle pour exhiber une base solide. À partir de cette formalisation nous pourrons en tirer une formalisation informatique qui profitera des démonstrations effectuées dans le cadre mathématique. La base formelle sera un élément essentiel du système REGAIN, successeur du système SACADO.

CONCLUSION.

Tout au long de ce mémoire, notre objectif a été de proposer un ensemble de modèles et d'outils pour aider au développement des interfaces utilisateur en C.F.A.O. Les outils proposés répondent aux problèmes que l'on rencontre couramment lors de la création d'une interface et décrits dans le chapitre 1 : complexité des outils existants et intégration insuffisante de l'utilisateur dans le développement. Le caractère générique des modèles d'interaction ou de dialogue proposés à l'heure actuelle est pénalisant dans la mesure où, le domaine d'application n'étant pas défini, ils n'aident que partiellement au développement. Ce travail a donné lieu à un important développement algorithmique nécessaire à la mise en œuvre des différents concepts présentés.

Le modèle d'interaction que nous proposons dans le chapitre 2 est issu d'un effort de formalisation pour aboutir à un système fiable et performant. La primitive unique de dialogue INTERACTION associée aux compatibilités et aux opérateurs de composition autorisent à la fois un contrôle des fils d'activités multiples de l'utilisateur (tâches-sous tâches) et une grande liberté. Toutes les tâches se font en accord avec la sémantique sans pour autant contraindre l'utilisateur dans un contexte limité. De plus, l'intérêt de ce modèle d'interaction est de proposer une solution de haut niveau pour décrire l'architecture d'un système autour du dialogue. La recherche de l'indépendance entre le dialogue et l'application pourra être renforcée dans le futur par une étude d'un modèle pour le composant *Présentation* de l'interface pour permettre une plus grande autonomie pour chacune des interactions. La maquette développée de SACADO (annexe A) intègre l'ensemble des fonctionnalités de notre modèle d'interaction et a permis de valider nos concepts sur le dialogue homme-machine en C.F.A.O.

Lechapitre 3 montre comment notre modèle d'interaction permet aux trois utilisateurs de SACADO d'intervenir dans le développement d'un système. Le langage NADRAG rapproche le prototype et le produit final. Il fournit un langage unique destiné à rapprocher le programmeur d'interfaces et le programmeur d'applications. Le développement du dialogue et de l'application se fait au même niveau ce qui permet une intégration souple et rapide. Le premier interpréteur réalisé ne comporte qu'un nombre retreint d'instructions pour l'application ce qui en limite son intérêt (annexe B). Des travaux futurs sur ce langage sont à envisager notamment sur la spécification des instructions dédiées au composant *Application*. Quant au langage graphique proposé, il est destiné plus particulièrement à des utilisateurs non informaticiens, il rapproche l'utilisateur du programmeur d'interfaces. Il est basé sur l'utilisation du modèle des réseaux de transition dont l'utilisation est très simple dans leur version de base, en apportant une réponse aux principales critiques que l'on peut lui faire grâce à notre modèle d'interaction. Le caractère global des réseaux est remplacé par une grande modularité et les états (modes) sont représentés par des interactions contextuelles. Ce langage graphique a permis la modification de la maquette SACADO (annexe C) et a servi de base pour les travaux du chapitre 4.

Dans le chapitre 4, nous mettons en évidence la dualité qui existe entre le modèle des objets et le développement du dialogue. Un grand nombre d'informations du modèle convenablement décrites peuvent être utilisées pour déduire un dialogue, dialogue qui peut être adapté aux besoins de

l'utilisateur. Cela nous garantit l'exhaustivité et la complétude des dialogues. Ainsi, nous avons proposé un modèle des objets décrit à l'aide de comportements qui ont un sens ou un lien avec le dialogue. À partir de la définition de ces comportements et de règles de génération, nous fournissons un prototype opérationnel qui peut servir de base de travail. La maquette décrite en annexe D a montré le bien-fondé de ce point de vue. En partant de la définition du modèles des entités manipulées par le langage graphique du chapitre 3, nous avons obtenu un outil hautement interactif. Les difficultés rencontrées dans ce chapitre proviennent essentiellement de la définition du modèle et de l'énoncé des règles de génération. La spécification du modèle par les comportements est un thème actuellement en cours de traitement dans notre laboratoire et nous pourrons alors nous appuyer sur une base plus formelle pour envisager le développement du système REGAIN, successeur du système SACADO. Les différentes générations que nous avons proposées ne sont que des exemples et nous envisageons de nombreuses évolutions qui mettront en valeur nos propositions car toute nouvelle règle enrichit automatiquement les implantations sans autre modification.

Enfin, nous envisageons d'étudier l'utilisation de notre modèle par les comportements dans d'autres cas de figure, et en particulier pour aborder le problème de la multi-modélisation et des modèles locaux. Les comportements peuvent fournir des informations très importantes pour déterminer les informations sémantiques à déléguer dans un modèle local (par analogie au modèle IDS). Quant aux interactions, elles fournissent des informations sur le moment de cette délégation. On pourra ainsi chercher à répondre à la principale difficulté du modèle IDS qui est de ne pas disposer d'un modèle des objets bien formalisé pour déterminer les informations à déléguer.

RÉFÉRENCES BIBLIOGRAPHIQUES.

[ABO 90] G. ABOWD, Agents : recognition and interaction models, in D. Diaper, D. Gilmore, G. Cockton et B. Shackel editors, *Human-Computer Interaction, Proceedings of INTERACT'90*, p. 143 à 146, North-Holland, Amsterdam, 1990.

[AHO 90] A. AHO, R. SETHI et J. ULLMAN, *COMPILATEURS, Principes, techniques et outils*, InterÉditions, Paris, 1990.

[ALP 93] S. R. ALPERT, Graceful Interaction with Graphical Constraints, *IEEE Computer Graphics & Applications*, Mars 1993, p. 82 à 91.

[ART 87] J. D. ARTHUR, Toward a Formal Specification of Menu-Based Systems, *The Journal of Systems and Software*, 1987, Vol. 7, p. 73 à 82.

[BAE 95] R. A. BAECKER, J. GRUDIN, W. A. S. BUXTON et S. GREENBERG, *Human-Computer Interaction : Toward the Year 2000*, Morgan Kaufmann Publishers, 1995.

[BAR 86] M. F. BARTHET et C. SIBERTIN-BLANC, La Modélisation d'Applications Interactives Adaptées aux Utilisateurs par des Réseaux de Pétri à Structure de Données, *Actes du Troisième Colloque-Exposition de Génie Logiciel*, AFCET, Mai1986, p. 117 à 136.

[BAR 89] L. A. BARFORD et B. T. VANDER ZANDEN, Attribute Grammars in Constraint-Based Graphics Systems, *Software-Practice and Experience*, 1989, Vol. 19, Num. 4, p. 309 à 328.

[BET 88] J. BETTELS, P. R. BONO, E. MCGINNIS et J. RIX, Guidelines for Determining When to Use GKS and When to Use PHIGS, *Computer Graphics Forum*, 1988, Num. 7, p. 347 à 354.

[BOO 91] G. BOOCH, *Object Oriented Design With Applications*, The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.

[BOR 92] M. BORDEGONI, U. CUGINI et C. RIZZI, A Visual Programming Tool for the Construction of Man-Machine Interface, *Proceedings of MICAD'92*, p. 129 à 143.

[BRU 93] H. DE BRUIN, P. BOUWMAN and J. VAN DEN BOS, DIGIS A Graphical User Interface Design Environment for Non-Programmers, *EuroGraphics'93*, 1993, Vol. 12, Num. 3, p. 13 à 24.

[CAI 92] S. CAI, W. LI et H. ZHANG, A Visual Tool for User-Interface Development, *The Visual Computer*, 1992, Vol. 8, Num. 2, p. 134 à 143.

[CHA 93] B. CHABRIER, *Interfaces par contraintes graphiques*, Thèse d'université, Nice - Sophia Antipolis, 7 Octobre 1993.

[CHU 88] Y. CHU, Petri Net Design Language, *IEEE*, 1988, p. 96 à 99.

[COU 90] J. COUTAZ, *Interfaces homme-ordinateur, Conception et réalisation*, DUNOD informatique, Paris, 1990.

[DAN 87] J. R. DANCE, T. E. GRANOR, R. D. HILL, S. E. HUDSON, J. MEADS, B. A. MYERS et A. SHULERT, The Run-Time Structure of UIMS-Supported Applications, *Computer Graphics*, 1987, Vol. 21, Num. 2, p. 97 à 101.

[DAV 89] R. DAVID et H. ALLA, *Du Grafcet aux réseaux de Petri*, Traité des nouvelles technologies, Hermès, 1989.

[DEN 92] J. M. DENNIS, J. DE BAAR, J. D. FOLEY et K. E. MULLET, Coupling Application Design and User Interface Design, in *Proceedings of the ACM Conference on Computer-Human Interaction*, 1992, ACM Press, New York, p. 259 à 266.

[DIN 93] D. DINGELDEIN, THESEUS++ : a high level user interface toolkit for graphical applications, *Computers & graphics*, 1993, Vol. 17, Num. 2, p. 147 à 154.

[DIX 93] A. DIX, J. FINLAY, G. ABOWD et R. BEALE, *Human-Computer Interaction*, Prentice-Hall editors, 1993.

[DUV 91] T. DUVAL, Interfaces Homme-Machine : Évaluation du Modèle d'Architecture Logicielle PAC, *Revue internationale de CFAO et d'infographie*, 1991, Vol. 6, Num. 2, p. 113 à 134.

[DUV 93] T. DUVAL, SPA (Seeheim-PAC-Arche) : un modèle d'architecture logicielle pour les applications interactives, *Actes de MICAD 93*, p. 245 à 264.

[DUV 94] T. DUVAL, Modèles d'architecture pour les applications graphiques interactives : la famille Seeheim, *Revue internationale de CFAO et d'infographie*, 1994, Vol. 9, Num. 4, p. 529 à 555.

[FAC 92] G. P. FACONTI et F. PATERNO, A Visual Environment to Define Composition of Interacting Graphical Objects, *The Visual Computer*, 1992, Vol. 9, p. 73 à 83.

[FIR 91] C. FIRTH et R. C. THOMAS, The Use of Command Language Grammar in a Design Tool, *International Journal of Man-Machine Studies*, 1991, Vol. 34, p. 479 à 496.

[FIS 90] D. L . FISHER, E. J. YUNGKURTH et S. M. MOSS, Optimal Menu Hierarchy Design : Syntax and Semantics, *Human Factors*, 1990, Vol. 32, Num. 6, p. 665 à 683.

[FOL 90] J. FOLEY, A. VAN DAM, S. FEINER et J. HUGUES, *Computer Graphics, Principles and Practice*, Addison-Wesley, 1990, Second Edition.

[FOL 93] J. D. FOLEY, P. SUKAVIRIYA et T. GRIFFITH, A Second Generation User Interface Design Environment : The Model and The Runtime Architecture, in *Proceedings of INTERCHI'93*, ACM Press, 24-29 Avril, 1993, p. 375 à 382.

[FRO 93] D. M. FROHLICH, The history and future of direct manipulation, *Behaviour & information technology*, 1993, Vol. 12, Num. 6, p. 315 à 329.

[GAR 88] Y. GARDAN, J.-P. JUNG, J.-N. KOLOPP, C. MINICH et W. TOTINO, Une approche nouvelle de la convivialité dans un système de C.A.O. : les principes du dialogue dans SACADO, *Actes MICAD 88*, Hermès, 21-25 Mars, 1988, p. 281 à 296.

[GAR 90] Y. GARDAN et J.-P. JUNG, A new kind of generators for CAD/CAM, *5ème CARs and FOF*, Norfolk (USA), December 1990.

[GAR 91] Y. GARDAN, *La CFAO introduction, techniques, et mise en œuvre*, Hermès, 3ème édition, 1991.

[GAR 92] Y. GARDAN, J.-P. JUNG, B. MARTIN et D. VOYAT, *NADRAG : formalisation des menus et du langage*, Rapport de recherche, Juillet 1992.

[GAR 93] Y. GARDAN, J.-P. JUNG et B. MARTIN, An End-User oriented approach to design man-machine interface for CAD/CAM, in *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, Le Touquet, FRANCE, 17-20 Octobre, 1993, p. 525 à 530.

[GAR 95a] Y. GARDAN, J.-P. JUNG, S. LEINEN, B. MARTIN, C. MINICH, C. POINSIGNON et I. STÉMART, *Conception et réalisation d'une maquette illustrant une approche du dialogue, de la gestion des contraintes et de la modélisation en C.A.O.*, Rapport de recherche, L.R.I.M., n°95-01, Mars 1995.

[GAR 95b] Y. GARDAN, B. MARTIN et I. STÉMART, *Modélisation des comportements : un lien entre modèles et dialogues*, Rapport de recherche N.95-02, LRIM, Metz, Avril 1995.

[GAR 96] Y. GARDAN, B. MARTIN et I. STÉMART, Extraction des connaissances du modèle en CFAO : application à la génération du dialogue, accepté pour *MICAD'96*, Paris (FRANCE), 13-15 Février 1996, à paraître dans la *Revue Internationale de CFAO et d'infographie* de février 1996.

[GIR 93] P. GIRARD et G. PIERRA, *End-user programming environments : a taxonomy revisited*, Rapport de recherche N°93009, Ecole Nationale Supérieure de Mécanique et d'Aérotechnique.

[GIE 92] D. F. GIESKENS and J. D. FOLEY, Controlling user interface objects through pre- and post-conditions, in *Proceedings of CHI'92*, ACM Press, May 3-7, 1992, p. 189 à 194.

[GOS 88] D. C. GOSSARD, R. P. ZUFFANTE and H. SAKURAI, Representing Dimensions, Tolerances, and Features in MCAE Systems, *IEEE Computer Graphics & Applications*, March 1988, p. 51-59.

[GRA 94] B. GRAU, G. SABAH et A. VILNAT, Pragmatique et dialogue homme-machine, *Technique et science informatiques*, Vol. 13, Num. 1, 1994, p. 9 à 30.

[GRE 86] M. GREEN, A Survey of Three Dialogue Models, *ACM Transactions on Graphics*, Juillet 1986, Vol. 5, Num. 3, p. 244 à 275.

[GUI 93] L. GUITTET et G. PIERRA, *Organisation modulaire et spécification d'une application de C.A.O. : Hiérarchie d'interacteurs dans le système NODAO*, Rapport de recherche , Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, n°93011, 1993.

[HAN 88] P. A. HANCOCK, Mental Workload Dynamic in Adaptative Interface Design, *IEEE Transactions on Systems, Man and Cybernetics*, Juillet/Août 1988, Vol. 18, Num. 4, p. 647 à 658.

[HAR 90] H. R. HARTSON, D. HIX et T. M. KRALY, Developing Human-Computer Interface Models and Representation Techniques, *IEEE Software-Practice and Experience*, Mai 1990, Vol. 20, Num. 5, p. 425 à 457.

[HAY 85] P. J. HAYES, P. A. SZEKELY et R. A. LERNER, Design Alternatives for User-Interface Management Systems Based on Experience with Cousin, *Proceedings of SIGCHI'85*, ACM Press, New York, 1985, p. 169 à 175.

[HAR 91] J. HARDENBERGH et J. MICHENER, Integrating PHIGS and User Interface Systems, *Computer Graphics Forum*, 1991, Vol. 10, p. 27 à 36.

[HIG 94] P. HIGUET, *Etude des prolongements d'un langage de description d'actions adapté à la C.A.O.*, Mémoire d'ingénieur C.N.A.M., L.R.I.M., 16 Mai 1994.

[HUB 89] W. HÜBNER et M. R. GOMES, Two Object-Oriented Models to Design Graphical User Interfaces, in *Proceedings of EUROGRAPHICS'89*, North-Holland, 1989, p. 63 à 74.

[JAC 86] R. J. K. JACOB, A Specification Language for Direct Manipulation Interfaces, *ACM Transactions on Graphics*, Octobre 1986, Vol. 5, Num. 4, p. 283 à 317.

[JAN 93] C. JANSSEN, A. WEISBECKER and J. ZIEGLER, Generating User Interfaces from Data models and Dialogue Net Specifications, in *Proceedings of INTERCHI'93*, ACM Press, 24-29 Avril, 1993, p. 418 à 423.

[JOH 93] J. A. JOHNSON, B. A. NARDI, C. L. ZARMER et J. R. MILLER, ACE : Building Interactive Graphical Applications, *Communications of the ACM*, Avril 1993, Vol. 36, Num. 4, p. 41 à 55.

[KAN 89] E. KANTOROWITZ et O. SUDARSKY, The Adaptable User Interface, *Communications of the ACM*, Novembre 1989, Vol. 32, Num. 11, p. 1352 à 1358.

[KAS 82] D. J. KASIK, A User Interface Management System, *Computer Graphics*, Juillet 1982, Vol. 16, Num. 3, p. 99 à 106.

[KIM 93] W. C. KIM et J. D. FOLEY, Providing High-level Control and Expert Assistance in the User Interface Presentation Design, in *Proceedings of INTERCHI'93*, ACM Press, 1993, April 24-29, p. 430 à 437.

[KOH 91] J.-M. KOHL, *Développement de générateurs de dialogue en CFAO*, Mémoire d'ingénieur CNAM, LRIM, Mai 1991.

[KUR 93] G. KURTENBACH and W. BUXTON, The Limits Of Expert Performance Using Hierarchic Marking Menus, in *Proceedings of INTERCHI'93*, ACM Press, 1993, April 24-29, p. 482 à 487.

[KUR 94] G. KURTENBACH and W. BUXTON, User Learning and Performance with Marking Menus, in *Proceedings of CHI'94, Human Factors in Computing Sciences*, Boston, Massachusetts, USA, 24-28 Avril, 1994, p. 258 à 264.

[LAH 95] J. LAHYANE, *Implantation d'un outil graphique de conception du dialogue et de l'interface du système de C.A.O. SACADO*, Rapport de stage de D.E.A., CRIN - LRIM, Septembre 1995.

[LEM 92] B. LEMAIRE, Construction d'explications : utilisation d'une architecture de tableau noir, *Actes des 4èmes Journées Nationales PRC-GDR Intelligence Artificielle*, Marseille (FRANCE), 19-21 Octobre, 1992, p. 471-494.

[MAR 91] A. MARCUS et A. VAN DAM, User-Interface Developments for the Nineties, *IEEE Computer*, Septembre 1991, Vol. 24, Num. 9, p. 49 à 57.

[MAR 92a] A. MARCUS, The Future of Advanced User Interfaces in Product Design, *IEEE*, 1992, p. 14 à 20.

[MAR 92b] B. MARTIN, *Contribution à la conception et à l'implémentation du langage NADRAG*, Rapport de stage de D.E.A., CRIN - LRIM, Septembre 1992.

[MAR 93] A. MARCUS, Human Communications Issues in Advanced UIs, *Communications of the ACM*, Avril 1993, Vol. 36, Num. 4, p. 101 à 109.

[MOR 93] A. MORSE et G. REYNOLDS, Overcoming Current Growth Limits in UI Development, *Communications of the ACM*, Avril 1993, Vol. 36, Num. 4, p. 73 à 81.

[MOR 94] R. MORIYON, P. SZEKELY et R. NECHES, Automatic Generation of Help from Interface Design Models, in *Proceedings of CHI'94, Human Factors in Computing Sciences*, Boston, Massachusetts, USA, 24-28 Avril, 1994, p. 225 à 231.

[MYE 89] B. A. MYERS, User-Interface Tools : Introduction and Survey, *IEEE Software*, Janvier 1989, Vol. 6, Num. 1, p. 15 à 23.

[MYE 90a] B. A. MYERS, Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints, *ACM Transactions on Programming Languages and Systems*, Avril 1990, Vol. 12, Num. 2, p. 143 à 177.

[MYE 90b] B. A. MYERS, Taxonomies of Visual Programming and Program Visualization, *Journal of Visual Languages and Computing*, Mars 1990, Vol. 1, Num. 1, p. 97 à 123.

[MYE 90c] B. A. MYERS, A New Model for Handling Input, *ACM Transactions on Information Systems*, Juillet 1990, Vol. 8, Num. 3, p. 289 à 320.

[MYE 91] B. A. MYERS, Graphical Techniques in a Spreadsheet for Specifying User Interfaces, *Human Factors in Computing Systems*, Avril 1991, New Orleans, LA, p. 243 à 249. Proceedings SIGCHI'91.

[MYE 92a] B. A. MYERS, Demonstrational Interfaces : A Step Beyond Direct Manipulation, *IEEE Computer*, Août 1992, p. 61 à 73.

[MYE 92b] B. A. MYERS, *Languages for Developing User Interfaces*, Boston : Jones and Bartlett, 1992.

[MYE 92c] B. A. MYERS et M. B. ROSSON, Survey on User Interface Programming, *Proceedings of the ACM Conference on Computer-Human Interaction*, 1992, ACM Press, New York, p. 195 à 202.

[MYE 92d] B. A. MYERS et B. VANDER ZANDEN, Environment for Rapidly Creating Interactive Design Tools, *The Visual Computer*, 1992, Vol. 8, Num. 2, p. 94 à 116.

[MYE 93] B. A. MYERS, R. G. MCDANIEL et D. S. KOSBIE, Marquise : Creating Complete User Interfaces by Demonstration, in *Proceedings of INTERCHI'93*, ACM Press, 1993, April 24-29, p. 293 à 300.

[NEE 91] F. NEELAMKAVIL et O. MULLARNEY, A Methodology and Tool Set for Supporting the Development of Graphical User Interfaces, *Computer Graphics Forum*, 1991, Vol. 10, p. 37 à 47.

[NOR 86] D. A. NORMAN et S. W. DRAPER, *User-Centred System Design : New Perspectives on Human-Computer Interaction*, Laurence Erlbaum Associates, New Jersey, 1986.

[OLS 92] D. OLSEN JR., T. MCNEILL et D. MICHEL, Workspaces : An architecture for editing collections of objects, in *Proceedings of the ACM Conference on Computer-Human Interaction*, ACM Press, New York, 1992, p. 267 à 272.

[PAN 93] G. J. PANGALOS, Designing the user interface, *Computers in Industry*, 1993, Num. 22, p. 193 à 200.

[PIE 87] J.-M. PIERREL, *Dialogue oral homme-machine*, Hermès, Paris, 1987.

[PIP 95] T. PIPIERI, *Implémentation d'un nouveau dialogue dans le système SACADO*, Mémoire d'ingénieur C.N.A.M., L.R.I.M., Février 1995.

[PRE 94] J. PREECE, Y. ROGERS, H. SHARP, D. BENYON, S. HOLLAND et T. CAREY, *Human-Computer Interaction*, Addison-Wesley, 1994.

[RUS 93] P. RUSSO et S. BOOR, How Fluent is Your Interface ? Designing for International Users, in *Proceedings of INTERCHI'93*, ACM Press, 24-29 Avril, 1993, p. 342 à 347.

[SAD 92] S. SADOU, *Interface utilisateur des systèmes interactifs complexes*, Thèse d'université, École centrale de Lyon, 23 Janvier 1992.

[SHE 92] T. SHEPARD, S. SIBBALD et C. WORTLEY, A Visual Software Process Language, *Communications of the ACM*, Février 1992, Vol. 35, Num. 4, p. 37 à 44.

[SHN 83] B. SHNEIDERMAN, Direct Manipulation : a step beyond programming languages, *Computer*, 1983, Vol. 16, Num. 8, p. 57 à 69.

[SHN 91] B. SHNEIDERMAN, Taxonomy and Rule Base for the Selection of Interaction Styles, in B. and S. RICHARDSON editors, *Human Factors for Informatics Usability*, Cambridge University Press, p. 325 à 342.

[SHO 90] P. SHOVAL, Functional Design of a Menu-Tree Interface within Structured System Development, *International Journal of Man-Machine Studies*, 1990, Vol. 33, p. 537 à 556.

[SIN 90] G. SINGH, Vu : Visual User-Interface Design, *The Visual Computer*, 1990, Vol. 6, p. 230 à 241.

[SLO 89] I. SLOMIANY, *Un langage interprété spécifique à la description de pièces paramétrées en C.A.O.*, Mémoire d'ingénieur C.N.A.M., L.R.I.M., Octobre 1989.

[SZE 89] P. SZEKELY, Standardizing the Interface Between Applications And UIMSs, in *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, ACM Press, New York, 1989, p. 34 à 42.

[SZE 93] P. SZEKELY, P. LUO and R. NECHES, Beyond interface builders : model-based interface tools, in *Proceedings of INTERCHI'93*, ACM Press, 24-29 Avril, 1993, p. 383 à 390.

[SZE 94] P. SZEKELY, R. MORIYON and R. NECHES, Automatic Generation of Help from Interface Design Models, in *Proceedings of CHI'94*, Boston, Massachusetts (USA), 24-28 Avril, 1994, pp. 225 à 231.

[TOT 89] W. TOTINO, *Contribution à la modélisation de l'interface homme-machine dans un système de CAO*, Thèse d'université, Metz, 6 Octobre 1989.

[VLI 90] J. M. VLISSIDES et M. A. LINTON, Unidraw : A framework for building domain-specific graphical editors, *ACM Transactions on Information Systems*, Vol. 8, Num. 3, Juillet 1990, p. 237 à 268.

[VOY 92] D. VOYAT, *Conception et implémentation d'un générateur de menus adapté à la C.A.O.*, Mémoire d'ingénieur C.N.A.M., L.R.I.M., 7 Décembre 1992.

[WAS 85] A. I. WASSERMAN, Extending State Transition Diagrams for the Specification of Human-Computer Interaction, *IEEE Transactions on Software Engineering*, Août 1985, Vol. 11, Num. 8, p. 699 à 713.

[WOL 91] J. WOLTER and P. CHANDRASEKARAN, A Concept for a Constraint-Based Representation of Functional and Geometric Design Knowledge, in *Proceedings Symposium on solid modeling fundation and CAD/CAM application*, Austin, Texas (USA), 5-7 Juin, 1991, p. 409 à 418.

[WOO 88] C. A. WOOD, P. D. GRAY et A. C. KILGOUR, Experience with Chisl, a Configurable Hierarchical Interface Specification Language, *Computer Graphics Forum*, 1988, Vol. 7, p. 117 à 127.

[ZEL 93] R. C. ZELEZNIK, K. P. HERNDON, D. C. ROBBINS, N. HUANG, T. MEYER, N. PARKER et J. F. HUGHES, An Interactive 3D Toolkit for Constructing 3D Widgets, *COMPUTER GRAPHICS Proceedings*, Annual Conference Series, 1993, p. 81 à 84.

ANNEXES.

LE LANGAGE D'IMPLÉMENTATION.

1. LE LANGAGE D'IMPLÉMENTATION.

Toutes les implémentations réalisées au cours de cette thèse l'ont été avec le langage C++, qui est un langage orienté objets. Le concept clé de C++ est la "classe". Une classe est un type défini par l'utilisateur décrit à la fois par les données qui le compose et par les méthodes agissant sur ces données. On appelle "instance" d'une classe un objet appartenant à cette classe. Pour construire tous les programmes, nous avons été amenés à définir une décomposition des problèmes en termes de classes. Pour cela, nous avons utilisé l'approche proposée par Grady BOOCH pour la conception orientée objets. Son approche est basée sur un formalisme qui reprend les quatre étapes classiques des méthodologies : spécification, analyse, conception et programmation. Seuls les étapes d'analyse et de conception nous ont servi. Nous présentons brièvement ce formalisme dans le paragraphe suivant à travers les notions que nous utiliserons dans les annexes. Pour une description plus complète, il est toujours possible de consulter [BOO 91].

2. LA MÉTHODE DE DÉVELOPPEMENT.

Le formalisme de Grady BOOCH est un formalisme graphique, orienté objets, qui permet la description des entités manipulées dans un projet, ainsi que des liens entre ces entités. La description d'un projet s'effectue par l'intermédiaire de différents diagrammes :

- le diagramme des classes. Il montre l'existence des classes et leurs relations. Il représente tout ou partie de la structure de classes d'un système. Si pour un petit système, un seul diagramme suffit, les systèmes plus importants en requièrent plusieurs.
- le diagramme des objets. Il montre l'existence des objets et leurs relations. Il représente tout ou partie de la structure des objets d'un système et illustre la sémantique des mécanismes clés de la conception;
- le diagramme des modules. Il montre l'allocation des classes et des objets aux modules dans la conception physique d'un système. Il permet de documenter l'organisation physique de l'application développée. Il est ainsi possible d'avoir des informations sur la façon dont les classes sont regroupées en modules compilables, et sur les modules à lier pour obtenir un code exécutable;
- le diagramme des processus. Il montre l'allocation des processus aux processeurs dans la conception physique d'un système. Pour chaque processeur, il est possible d'indiquer les processus qui s'exécutent. Ce type de diagramme n'est utilisé que dans le cadre de développement utilisant le parallélisme.

Les deux premiers font partie de la vue logique d'un système, car ils servent à décrire l'existence et le sens des abstractions de la conception. Les deux autres font partie de la vue physique d'un système, car ils sont utilisés pour décrire les composants logiciels et matériels d'une implantation.

Ces quatre diagrammes font partie de ce que l'on appelle la sémantique statique. En conception orientée objets, la sémantique dynamique s'exprime à travers deux diagrammes additionnels :

- le diagramme des états. Il montre l'état d'une instance d'une classe donnée, les événements qui causent une transition d'un état à un autre, et les actions qui résultent de ce changement d'état. Il y a trois sortes d'états : l'état de départ, les états intermédiaires et l'état final.
- le diagramme des temps. Il documente le séquencement des opérations en indiquant les interactions dynamiques entre divers objets dans un diagramme des objets.

Dans les paragraphes suivants, Nous détaillons essentiellement le diagramme des classes et le diagramme des objets, diagrammes que nous avons utilisé pour préciser nos choix d'implantations.

2.1. Le diagramme des classes.

Le diagramme des classes utilise trois types d'éléments de base :

- les classes;
- les relations entre classes;
- les classes utilitaires.

2.1.1. Les classes.

Elle est représentée par un nuage contenant le nom de la classe :

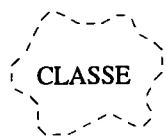


Figure 1. *Une classe.*

2.1.2. Les relations.

Il existe différentes liaisons possibles entre classes. Nous présentons exclusivement les deux types de liaisons que nous utilisons dans les annexes. Nous pouvons remarquer que la première liaison possède une indication de cardinalité représentée par le caractère "x" :

Figure 2. *Les liaisons entre classes.*

Les cardinalités utilisables sont 0 (zéro), 1 (un), * (zéro ou plus), + (un ou plus), ? (zéro ou un) et n (n). Nous donnons un exemple pour chacune de ces deux liaisons :

- La liaison "utilise" entre les classes A et B (figure 2.6) indique que la classe A utilise la classe B (par exemple, un attribut de type liste de B y est déclaré). De plus, les cardinalités indiquent que chaque instance de B est en relation avec une instance de A, et que chaque instance de A est en relation avec une ou plusieurs instances de B :

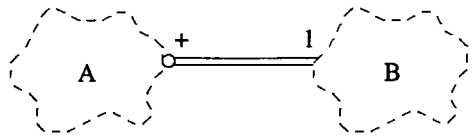


Figure 3. *La liaison "utilise"*

- La liaison "hérite" entre les classes A et B (figure 2.7) indique que la classe A possède toutes les caractéristiques de la classe B qui est dite plus générale. La classe A hérite des attributs et des services de la classe B :

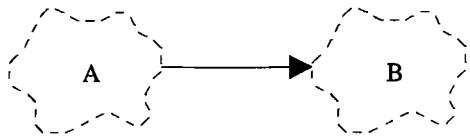


Figure 4. *La liaison "hérite".*

Une classe utilitaire est un regroupement de services qui ne sont pas liés à un type de données particulier (assimilable à une bibliothèque de fonctions). Elle est représentée par un nuage ombré contenant le nom de la classe utilitaire :

Figure 5. *La classe utilitaire.*

2.2. Le diagramme des objets.

Rappelons qu'un objet est une instance de classe. Le diagramme des objets est utilisé pour montrer l'existence des objets et leurs relations. Il représente tout ou partie de la structure d'objets d'un système et illustre la sémantique des mécanismes clés de la conception.

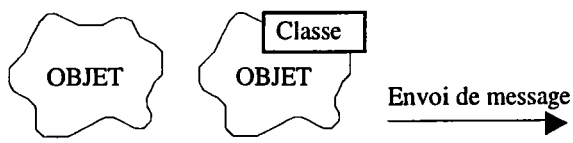


Figure 6. *Un objet.*

Un objet est représenté par un nuage en trait plein (à l'intérieur du nuage figure le nom de l'objet). Par extension aux notations de Grady BOOCH, on peut faire figurer en plus dans un rectangle le nom de la classe. Le seul type de relation entre objets est l'envoi de messages pour demander l'exécution d'un service. Cette relation est matérialisée par une flèche simple (les diverses formes de cette relation ne sont pas détaillées).

2.3. Exemple.

Nous montrons un exemple de diagramme que l'on peut obtenir en utilisant les classes et les liaisons "utilise" et "hérite". Cet exemple propose de définir une décomposition en classes des véhicules "camions", "voitures" et "motos" à travers une de leur composante, la roue :

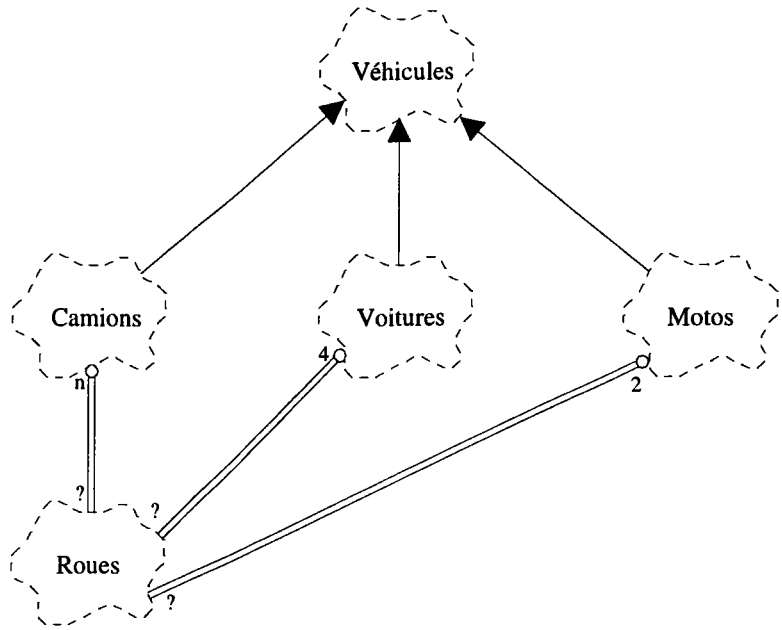


Figure 7. *Un exemple de diagramme des classes.*

Cette figure montre une hiérarchie d'héritage car les classes "Camions", "Voitures" et "Motos" héritent de la classe "Véhicules", ainsi qu'une composition car les relations "utilise" montrent qu'un camion peut avoir n roues, qu'une voiture possède quatre roues et qu'une moto en a deux (généralement !).

ANNEXE A.

LE DÉVELOPPEMENT DE SACADO.

1. INTRODUCTION.

Comme nous l'avons vu dans le chapitre 2 de ce mémoire, SACADO est à la fois un outil de CAO et une base de développement. Nous appelons *noyau* cette base de développement pour la suite de cette annexe. L'objectif principal de ce noyau est de satisfaire au mieux les spécifications énoncées dans les chapitres précédentes (menus, compatibilités, actions globales, ...).

2. ARCHITECTURE GLOBALE DE SACADO.

L'architecture de SACADO s'articule autour de trois grands composants :

- MODÈLES : ce composant représente le(s) modèle(s) de CAO utilisé(s). Nous avons défini ces modèles dans le chapitre 2 (modèles génériques, géométriques, ...). Pour ce noyau, nous avons utilisé un modèle 2D existant au laboratoire;
- ACTIONS : ce composant permet la création des fonctionnalités de base de l'application, c'est-à-dire les actions globales attachées aux menus;
- INTERFACES : ce composant est responsable du déclenchement et de la gestion des actions, ainsi que de la manipulation des différents éléments d'interface tels que les menus, les fenêtres, ...

La figure suivante illustre cette décomposition.

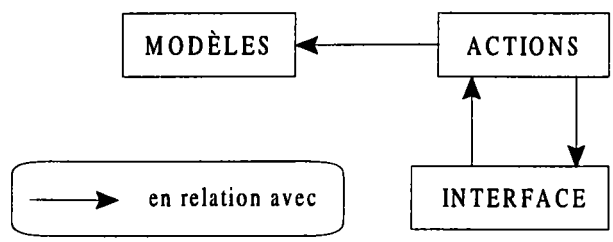


Figure A.1. *Architecture globale de SACADO.*

Dans les paragraphes suivants, nous allons détailler les composants INTERFACE et ACTIONS en indiquant pour chacun d'eux un diagramme des classes ainsi que les relations mises en place.

3. L'INTERFACE.

Le composant INTERFACE est responsable de la gestion de l'ensemble des éléments d'interface, du déclenchement des actions globales, du contrôle des entrées de l'opérateur et du respect des concepts du modèle d'interaction de SACADO.

```
class Interfaces
{
          char                                  *PtNom;
          Listes <ElementsInterface *>          ListeDesElements;
          Listes <ActionsGlobales *>            ListeDesActions;
          Piles  <ActionsEnCours *>             Pile;
public:
          méthodes ...
};
```

La classe *Interfaces* contient les fonctionnalites que l'on attend de la part de l'interface. Elle est définie par un nom (PtNom), par l'ensemble des éléments d'interface à considérer (ListeDesElements), par l'ensemble des actions globales de l'implantation considérée (ListeDesActions) et bien entendu par le contexte d'exécution chargé en particulier du maintient des fils d'activités multiples (Pile). Pour la mise à jour du contexte d'exécution, on peut se reporter au paragraphe 5. Les méthodes fournissent des mécanismes de construction et de manipulation de l'interface que nous étudions en détail.

3.1. Les éléments d'interface.

Les éléments d'interface regroupent tous les objets avec lesquels l'opérateur interagit. Ils définissent la partie statique de l'interface SACADO. On trouve notamment les menus, les fenêtres 2D, les zones de saisie scalaire et texte, les zones de message, les groupes, les domaines et les fenêtres d'application.

3.1.1. La structure utilisée.

Tous ces éléments d'interfaces ont été développés en deux parties : une partie indépendante de l'implantation qui contient des règles (vérification de la cohérence), des structures diverses (fenêtres, clôtures, fontes, ...) et une partie adaptation faisant le lien avec le système, en particulier avec le système à base de fenêtres choisi. Cette dernière couche permet à l'application d'être indépendante par rapport à la boîte à outils choisie (OLIT sur stations SUN). Pour garantir la séparation, nous avons décidé de décrire tous les éléments d'interface en deux classes distinctes, l'élément et son aspect.

Un aspect représente la partie visible par l'opérateur, c'est l'objet avec lequel l'opérateur interagit. La classe correspondante décrit l'ensemble des fonctionnalités que l'on attend de chacun des aspects. On retrouve un aspect par type d'élément d'interface mais il existe une classe de base, la classe *Aspects*.

```
class Aspects
{
        Gadgets *PtGadget;
public:
        ... méthodes de gestion des événements
};
```

Cette classe contient une référence vers l'objet de la bibliothèque qui sert d'aspect (PtGadget). Les méthodes couvrent la gestion des interactions avec l'opérateur (événements). À partir de la définition des aspects, on représente un élément d'interface par la classe *ElementsInterface*.

```
class ElementsInterface
{
        class ElementsListesAspects
        {
                Aspects *PtAspect;
        public:
                ... méthodes de manipulation des aspects
        };
        char                                    *PtNom;
        ActionsGlobales                         *ActionGlobale;
        Listes <ElementsListesAspects *>        ListeAspect;
public:
        ... méthodes de manipulation des éléments d'interface
};
```

Un élément d'interface est caractérisé par son nom (PtNom) et par l'action globale qui lui est associée. Nous considérons une liste d'aspects pour un élément donné. Cela permet à un même élément d'interface d'apparaître à plusieurs endroits (par exemple un menu est également présent dans une barre d'outils et les modifications de compatibilités du menu sont répercutées sur les deux aspects).

3.1.2. Les compatibilités.

Une compatibilité représente l'ensemble des modifications à apporter aux éléments d'interface lorsqu'une opération de l'opérateur est effectuée (sélection d'un menu, ...).

```
class Compatibilites
{
        class ElementsCompatibilite
        {
                CodesCompatibilite Code;
                ElementsInterface *PtElement;
        public:
                ... méthodes de manipulation
        };
        Listes <ElementsCompatibilite *>    Liste;
public:
        ... méthodes de manipulation
};
```

Une compatibilité est défini par une liste (Liste) qui contient un ensemble d'éléments d'interface (PtElement) avec pour chaque élément sa nouvelle compatibilité (Code).

3.1.3. La structuration.

Les éléments d'interface sont décrits par une structure hiérarchique et on distingue six classes dérivées de la classe de base *ElementsInterface* :

- un élément est dépendant ou indépendant (*ElementsInterfaceDépendants* et *ElementsInterfaceIndépendants*). Par exemple, une fenêtre d'application est indépendante des autres éléments d'interface;
- un élément possède ou non une compatibilité (*ElementsInterfaceDépendantsAvecCompatibilités* et *ElementsInterfaceIndépendantsAvecCompatibilités*). Par exemple un menu en possède une;
- un élément possède ou non des fils (*ElementsInterfaceDépendantsComplets* et *ElementsInterfaceIndépendantsComplets*). Par exemple un menu peut en posséder.

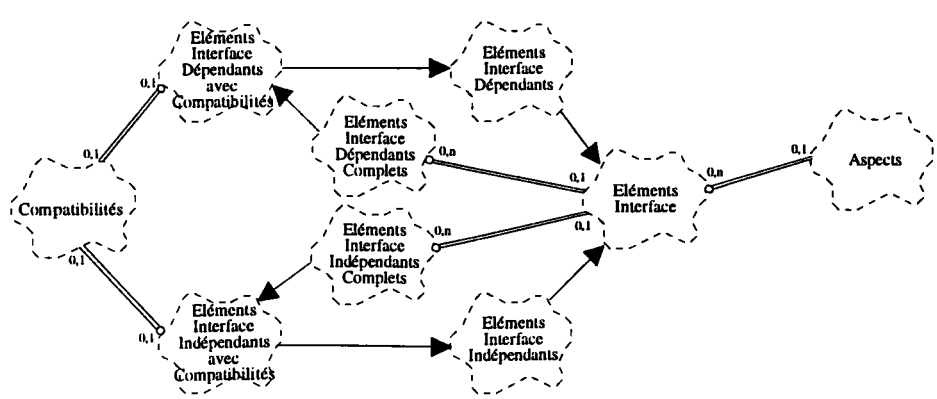


Figure A.2. *Le diagramme des classes des éléments d'interface.*

3.1.3.1. Les domaines.

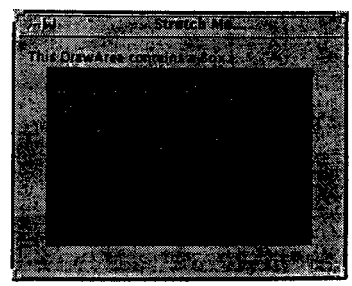
Un domaine est un élément d'interface indépendant complet. Il n'a pas de parent défini (indépendant), il possède une compatibilité dont ses fils héritent.

```
class Domaines:
 public ElementsInterfaceIndependantsComplets
{
public:
          ... méthodes de construction
};
```

Cet élément ne possède pas d'aspect, il est seulement utilisé pour regrouper des éléments d'interface de manière sémantique.

3.1.3.2. Les fenêtres d'application.

Cet élément d'interface est le premier élément avec lequel l'opérateur interagit. C'est la fenêtre visible qui définit l'application pour l'opérateur. Il dépend des domaines et possède une compatibilité dont ses fils héritent. Tous les autres éléments d'interface en sont issus (ils doivent appartenir à une fenêtre d'application).



```
class FenetresInterface:
 public ElementsInterfaceDependantsComplets
{
          FenetresGraphiques *Fen;
public:
          ... méthodes de manipulation
} ;
```

En dehors des méthodes de création et de manipulation, cet élément d'interface est défini par son aspect (Fen).

3.1.3.3. Les menus.

Les menus représentent les éléments d'interface avec lesquels l'opérateur choisit les opérations qu'il désire réaliser. Un menu peut posséder plusieurs fils et une compatibilité qui définit le contexte mis en place si l'opérateur le sélectionne. Un menu est dépendant, il ne peut pas exister sans appartenir à un autre élément d'interface.


```
class OptionsMenus:
 public ElementsInterfaceDependantsComplets
{
          Motlong    PosX, PosY;
public:
          ... méthodes de manipulation
};
```

Un menu est simplement précisé par sa position (PosX et PosY), position inutile si on utilise les menus dans des groupes. L'aspect des menus diffère selon que le menu possède ou non des fils (*AspectsMenuNonTerminal* et *AspectsMenuTerminal*).

3.1.3.4. Les groupes.

Les groupes sont des éléments d'interface qui permettent de grouper des éléments d'interface. Le placement est alors automatique (horizontal). C'est une simple commodité de création.

```
class GroupesDeMenus:
  public ElementsInterfaceDependantsComplets
{
            Motlong    PosX, PosY;
public:
            ... méthodes de manipulation
};
```

L'aspect de cet élément (*AspectsGroupe*) définit simplement cette opération par rapport à la boîte à outils. Les éléments d'interface groupés apparaissent avec un cadre noir qui les entoure.

3.1.3.5. Les zones d'affichage de texte.

Ces éléments d'interface autorisent l'application à informer l'opérateur des opérations réalisées grâce à des affichages particuliers.


```
class AffichagesTexte:
  public ElementsInterfaceDependants
{
public:
            ... méthodes de manipulation
};
```

L'aspect de cet élément (*AspectsAffichagesTexte*) est simplement défini par son positionnement et il est représenté par une zone non éditable où l'application peut afficher un texte quelconque.

3.1.3.6. Les zones de saisie.

Les entrées sont essentielles dans les systèmes interactifs. Nous avons défini deux styles de saisie, la saisie d'un texte et la saisie d'un nombre, styles qui dérivent d'une classe de base, la classe *Saisies*.

3.1.3.6.1. Les zones de saisie de texte.

Ces éléments d'interface fournissent une zone de texte éditable sur une seule ligne précédée d'un nom.


```
class SaisiesChaines:
  public Saisies
{

public:
            ... méthodes de manipulation
};
```

L'aspect de cet élément dépend essentiellement de son nom qui apparaît devant la zone de saisie.

3.1.3.6.2. Les zones de saisie de réel.

Ces éléments d'interface fournissent une zone numérique éditable précédée d'un nom. La valeur peut être tapée dans la zone ou modifiée par les flèches.

```
class SaisiesReels:
 public Saisies
{
public:
          ... méthodes de manipulations
};
```

L'aspect de cet élément dépend essentiellement de son nom qui apparaît devant la zone de saisie.

3.2. Le logiciel graphique de base.

Généralement, lorsque l'on développe une application, on recherche une certaine indépendance par rapport à l'ordinateur hôte, aux langages utilisés, aux terminaux graphiques. Le problème posé est de définir une "couche" au-dessus des moyens de visualisation et de dialogue qui permettrait de rendre le système de C.F.A.O. indépendant de ces moyens matériels. Cette "couche" est appelée LGB (Logiciel Graphique de Base). Le LGB comporte une interface avec le programme d'application (constituée de fonctions ou de primitives) indépendante du matériel, et une interface avec le terminal (gérant les moyens réellement disponibles) dépendante du matériel [GAR 91].

Le LGB s'occupe essentiellement des affichages entre l'espace opérateur et l'espace écran. L'espace opérateur représente le monde virtuel de l'opérateur (coordonnées exprimées en réels) alors que l'espace écran représente l'écran du terminal (coordonnées exprimées en entiers). On définit une fenêtre comme une vue sur un espace opérateur et une clôture comme une représentation visuelle d'une fenêtre dans une portion de l'espace écran.

Les objets affichés dans une clôture sont obtenus par localisation (passage fenêtre/clôture) et les objets manipulés par l'opérateur sont identifiés par globalisation (passage clôture/fenêtre).

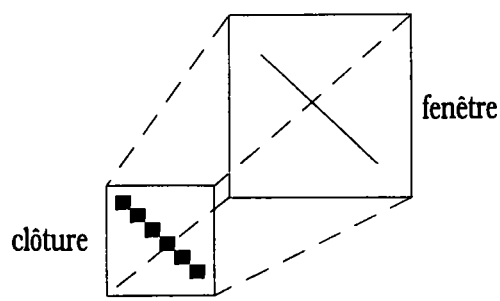


Figure A.3. *Association fenêtre/clôture (espace opérateur/espace écran).*

L'utilisation d'un langage orienté objets nous permet de regrouper les services offerts par le LGB vers l'application dans deux classes distinctes : *Clotures* et *Fenetres2D* (une instance de *Clotures* est toujours associée à une instance de *Fenetres2D*). Ces deux classes forment l'interface du LGB par rapport à l'application.

3.2.1. Les clôtures.

La classe *Clotures* représente l'espace écran et comprend une méthode de création (fenêtre 2D associée et taille), des méthodes de changement du style de tracé (couleur, ...), des méthodes de tracé (effacement, points, segments, texte, ...) et des méthodes de manipulation (déplacement, changement de taille, ...).

```
class Clotures
{
        ZonesDeDessin    *PtZoneDeDessin;
public:
        Clotures   (ElementsInterface   *Fen,   Motlong   Xbg,   Motlong   Ybg,
                    Motlong L, Motlong H);
        ... méthodes de changement de contexte
        Motlong CouleurTrace    (Motlong Couleur);
        ... méthodes de tracé
        void EffacerContenu ();
        Booleen Point (Motlong X, Motlong Y);
        ... méthodes de modification
        void   ModifierTaille   (Motlong   Xbg,   Motlong   Ybg,   Motlong   L,
                                  Motlong H);
};
```

Une clôture est définie par ses dimensions (Xbg, Ybg, L et H). Ces paramètres permettent de définir son aspect (PtZoneDeDessin, c'est un cadre dans lequel l'application peut dessiner). Un paramètre très important est représenté par la fenêtre à laquelle est associée la clôture (Fen). En effet, dans notre implantation, nous avons choisi de ne pas considérer de clôtures sans fenêtre, toute création se fera donc par l'intermédiaire d'une fenêtre (la fenêtre crée la clôture requise).

3.2.2. Les fenêtres.

La classe *Fenetres2D* est donc l'élément principal du LGB. C'est elle qui est responsable des passages fenêtre/clôture et inversement.

```
class Fenetres2D:
  public ElementsInterfaceDependants
{
        Clotures            *Clo;
public:
        Fenetres2D  (char  *Nom,  Motlong  TypeScene,  Reel  XFen,  Reel  YFen,
                     Reel  LFen,  Reel  HFen,  Motlong  XClo,  Motlong  YClo,
                     Motlong LClo, Motlong HClo);
        ... méthodes de tracé
        Booleen Point (Reel X, Reel Y);
};
```

Cet élément d'interface ne comporte pas d'aspect mais comprend une référence sur la clôture à laquelle il est associé (Clo). Lors de sa création, on indique les paramètres de la fenêtre (XFen, YFen, LFen et HFen) mais aussi les paramètres de sa représentation écran qui est la clôture (XClo, YClo, LClo et HClo). Pour un tracé en coordonnées réelles, il faut transformer les coordonnées en entiers et appeler les méthodes correspondantes de la clôture (CoeffFenClo et CoeffCloFen).

3.3. Les événements.

L'interface doit pouvoir traiter les événements qui arrivent de la boîte à outils (sélection d'un menu, désignation d'un objet dans une fenêtre 2D, ...). Nous disposons d'un module d'adaptation d'événements qui permet de traduire un événement X11 en un événement compréhensible par l'interface. Cette traduction nous garantit l'indépendance par rapport aux types d'événements.

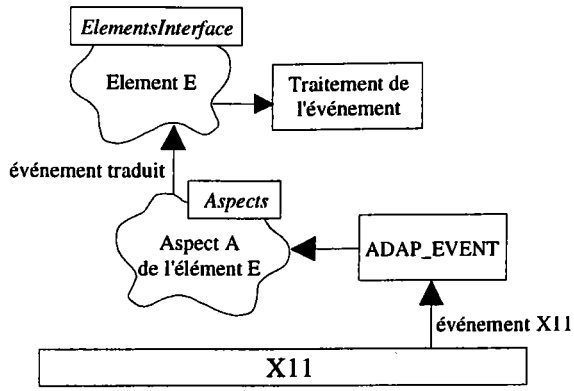


Figure A.4. *Le traitement des événements.*

Le traitement d'un événement se fait donc selon plusieurs étapes :

- l'événement X11 est traduit dans le module ADAP_EVENT;
- ce module informe l'aspect concerné (événement traduit);
- l'aspect prévient l'élément d'interface auquel il appartient;
- l'élément traite l'événement selon son type.

Par exemple, après sélection d'un menu (classe *OptionsMenus*), l'événement est envoyé à l'aspect qui en informe le menu correspondant. L'action globale associée à ce menu est alors exécutée.

3.4. Conclusion.

La hiérarchie des éléments d'interface ainsi que la définition des compatibilités nous permettent de construire la partie statique de l'interface par simple appel aux méthodes de construction des différents objets. Comme exemple, nous prenons l'interface déjà présentée dans la figure II.10 de ce mémoire.

```
Interfaces *InterfaceSimple; // déclaration de l'interface

InterfaceSimple = new Interfaces ("Exemple"); // création de l'interface
InterfaceSimple->CreerElement (new Domaines ("Domaine Principal"));
InterfaceSimple->MemoriserElement ( );
InterfaceSimple->CreerElement (new FenetresInterface ("Fenetre principale Sacado", "Sacado", 100, 1000, 100, 800,
        N, 300, 250, 600, 580);
InterfaceSimple->InsererDansElement ( );
InterfaceSimple->MemoriserElement ( );
InterfaceSimple->CreerElement (new Fenetres2D ("Fenetre 2D", 1, -1000.0, -1000.0, 2000.0, 2000.0, 10, 50, 500,
        500));
InterfaceSimple->InsererDansElement ( );
InterfaceSimple->CreerElement (new AffichagesTexte ("Message", 0, -1, -1, 20));
InterfaceSimple->InsererDansElement ( );
InterfaceSimple->CreerElement (new GroupesDeMenus ("Groupe");
InterfaceSimple->InsererDansElement ( );
InterfaceSimple->MemoriserElement ( );
```

```
InterfaceSimple->CreerElement (new OptionsMenus ("Quitter");
InterfaceSimple->InsererDansElement ( );
InterfaceSimple->CreerElement (new OptionsMenus ("Segment");
InterfaceSimple->InsererDansElement ( );
InterfaceSimple->CreerElement (new OptionsMenus ("Cercle");
InterfaceSimple->InsererDansElement ( );
InterfaceSimple->CreerElement (new OptionsMenus ("Point");
InterfaceSimple->InsererDansElement ( );
InterfaceSimple->MemoriserElement ( );
InterfaceSimple->CreerElement (new OptionsMenus ("Coordonnees");
InterfaceSimple->InsererDansElement ( );
InterfaceSimple->CreerElement (new OptionsMenus ("Intersection");
InterfaceSimple->InsererDansElement ( );
```

La méthode *CreerElement (...)* permet de créer un élément d'interface. Cette méthode affecte un champ "élément courant" contenant cet élément. *MemoriserElement ( )* permet de mémoriser l'élément courant dans un champ "élément mémorisé". *InsererElement ( )* insère l'élément courant dans l'élément mémorisé, c'est-à-dire que l'élément courant sera déclaré comme fils de l'élément mémorisé. Ces méthodes permettent de créer l'arborescence des éléments d'interfaces. Le résultat est la figure A.5.

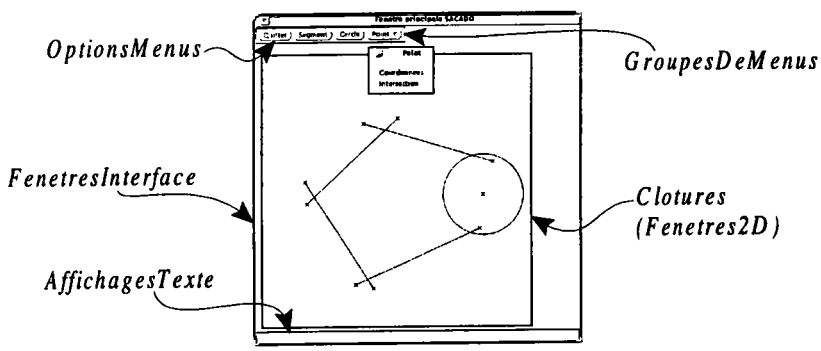


Figure A.5. *Exemple d'interface SACADO.*

On peut également déclarer les compatibilités entre les éléments d'interface :

```
InterfaceSimple->ChercherElement ("Cercle");
InterfaceSimple->MemoriserElement ( );
InterfaceSimple->ChercherElement ("Segment");
InterfaceSimple->InsererCompatibilité (Differe);
```

La méthode *ChercherElement (...)* permet de retrouver un élément créé précédemment et le met dans l'élément courant. *InsererCompatibilite (Immediat)* déclare que l'élément courant (donc celui précédemment cherché) est immédiat de l'élément mémorisé. Les compatibilités que l'on peut utiliser sont : immédiat, local, différé, inactif. Ainsi, pour la portion de code ci-dessus, le menu "Segment" est déclaré différé du menu "Cercle".

4. LES ACTIONS GLOBALES.

Les actions globales représentent les fonctionnalités de l'application et sont décrites par un graphe d'actions. Une action est soit une interaction, soit une action non interactive. On dit qu'une action globale est interactive si elle comporte au moins une interaction. Dans le cas contraire, elle est dite non interactive.

4.1. Description externe.

La description de la classe *ActionsGlobales* passe par l'analyse des besoins externes en termes de méthodes. La figure A.6. décrit une action globale à travers les objets qu'elle manipule.

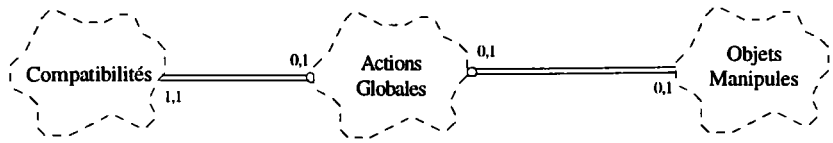


Figure A.6. *La description externe d'une action globale.*

Une action globale comporte une compatibilité de départ. Elle définit les compatibilités à appliquer lors du lancement de l'action. La classe *ObjetsManipules* représente les objets que l'on rencontre dans l'interface et le résultat de l'action globale en particulier.

```
class ObjetsManipules
{
          Motlong    LeType;
public:
          ... méthodes de manipulation
};
```

Cette classe de base décrit les données et les méthodes communes à tous les objets. On trouve essentiellement le type de l'objet (LeType). Comme objets que l'on peut rencontrer, on peut citer les coordonnées (*ObjetsCoordonnees*).

```
class ObjetsCoordonnees:
  public ObjetsManipules
{
          Motlong Xe, Ye;
          Clotures *PtCloture;
public:
          ObjetsCoordonnees (Motlong Xp, Motlong Yp, Clotures *Cloture);
          ... méthodes de manipulation
};
```

Comme on peut le constater, cette classe est définie par le couple de coordonnées (Xe, Ye) mais également par la clôture dans laquelle elles ont été saisies (PtCloture). Bien entendu, on trouve d'autres classes d'objets : réels, chaînes, ...

Pour mieux comprendre les actions globales, nous décrivons les fonctionnalités attendues en introduisant les données et les méthodes nécessaires :

- une action globale doit être activée. C'est le rôle de la méthode *Activer ()* qui amorce l'exécution de l'action globale (PremiereAction est la première action). Si l'action globale est non interactive, elle termine. Si l'action globale est interactive, elle se met en attente sur la première interaction rencontrée;
- une action globale doit pouvoir être réactivée lorsqu'elle est en attente. C'est le rôle de la méthode *Reactiver (objet)* qui reprend l'action globale où elle en était (ActionCourante est l'interaction en attente). L'objet en paramètre doit permettre de répondre à l'interaction en attente et l'exécution continue comme pour l'activation;

- la dernière interaction répondue doit pouvoir être annulée. C'est le rôle de la méthode *Annuler ()* qui remonte à l'interaction précédente;
- une action globale doit pouvoir être arrêtée. C'est le rôle de la méthode *Desactiver ()* qui stoppe définitivement l'action globale;
- une action globale doit pouvoir fournir son résultat. C'est le rôle de la méthode *DonnerValeur ()* qui demande son résultat à l'action contenant le résultat de l'action globale (ActionResultat représente cette action).

```
class ActionsGlobales
{
            char                *PtNom;
            Compatibilites      Compatibilite;
            Actions             *PremiereAction;
            Actions             *ActionResultat;
            Actions             *ActionCourante;
public:
            ObjetsManipules     *DonnerValeur   ();
            void                Activer         ();
            Booleen             Reactiver       (ObjetsManipules *ObjetManipule);
            Booleen             Annuler         ();
            void                Desactiver      ();
            ... méthodes de manipulation
};
```

En plus des méthodes et des données précédentes, une action globale est caractérisée par son nom (PtNom) et par sa compatibilité de départ (Compatibilite). Nous définissons la classe *Actions*.

4.2. Description interne.

On retrouve toutes les entités de notre modèle d'interaction (la primitive INTERACTION, les opérateurs de composition, les actions non interactives, ...).

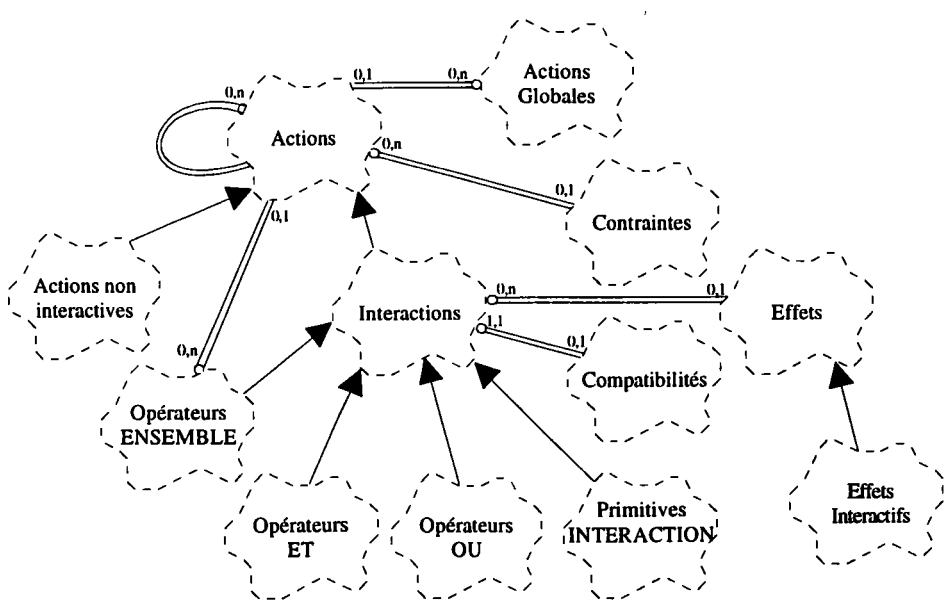


Figure A.7. *Le diagramme des classes des actions globales.*

Les actions non interactives, les effets et les contraintes sont représentés par des codes qui font référence à des primitives de l'application (fonctions ou procédures). Les contraintes permettent de vérifier le résultat d'une action (dans le cas des interactions, il s'agit de spécifier le composant *contrôle*

en précisant les objets demandés à l'opérateur, on parle de domaine restreint). Toutes les instances d'actions sont chaînées pour former le graphe de l'action globale désirée.

4.3. Conclusion.

La définition des actions globales nous permet une construction par simple appel aux méthodes des différentes classes. Comme exemple, nous prenons la création de l'action globale *ActGlob* de création d'un segment par deux points quelconques.

```
// ❶ déclaration de tous les objets utilisés dans l'action globale
Variables                              *VarInter1, *VarInter2, *VarAct;
ActionsNonInteractivesAvecResultat     *Act;
Interactions                           *Inter1, *Inter2;
ElementsInterface                      *Elt;

// ❷ instanciation des objets utilisés (variables, interactions, actions non interactives)
VarInter1 = new Variables ("Resultat Inter 1");
VarInter2 = new Variables ("Resultat Inter 2");
VarAct = new Variables ("Resultat Act");
Act = new ActionsNonInteractivesAvecResultat ("créer segment", 25, 2);
Inter1 = new Interactions ("Premier point", TypePoints);
Inter2 = new Interactions ("Deuxieme point", TypePoints);

// ❸ définition des compatibilités au niveau des interactions
Elt = INT_DonnerElement ("Point");
// sur la première interaction le menu "Point" est déclaré local
Inter1->Inserer (Local, Elt);
// sur la deuxième interaction le menu "Point" est déclaré local
Inter2->Inserer (Local, Elt);

// ❹ définition du séquencement de l'action globale
Inter1->AssocierActionSuivante (Inter2);
Inter2->AssocierActionSuivante (Act);

// ❺ récupération des résultats des interactions et des actions
Inter1->FixerResultat (VarInter1);
Inter2->FixerResultat (VarInter2);
Act->FixerResultat (VarAct);
// ❻ ajout des paramètres à l'action globale
Act->AjouterParametre (new ExpressionsVariable (VarInter1));
Act->AjouterParametre (new ExpressionsVariable (VarInter2));

// ❼ départ du graphe
ActGlob->PremiereAction = Inter1;
ActGlob->PtVariableResultat = VarAct;
```

On associe ensuite cette action au menu correspondant (on reprend l'interface créée précédemment).

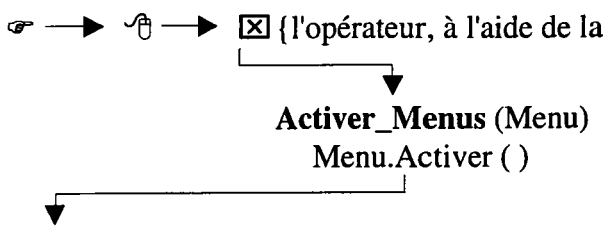
```
InterfaceSimple->ChercherElement ("Segment");
... création de l'action globale ActGlob ...
InterfaceSimple->AssocierAction (ActGlob);
```

La méthode *AssocierAction (action)* associe l'action globale *action* à l'élément courant. Les actions globales sont instanciées une fois à l'initialisation du système, comme pour les menus. Ainsi, dans la

portion de code ci-dessus, on recherche le menu "Segment" et on le stocke dans l'élément courant. On crée ensuite l'action globale "ActGlob". Enfin, on l'associe à l'élément courant.

5. LE FONCTIONNEMENT DE SACADO.

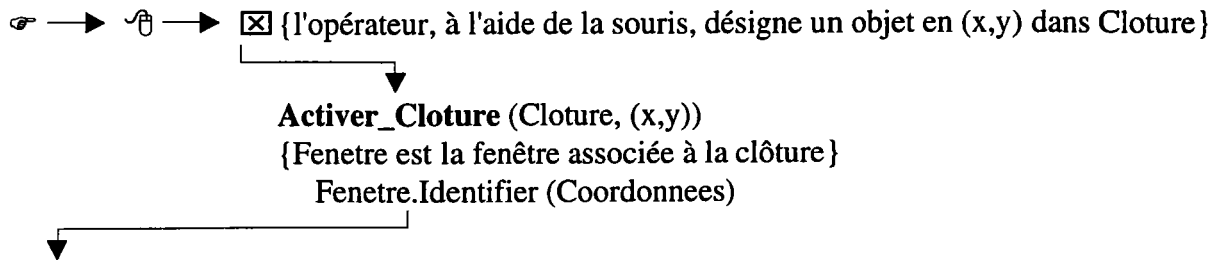
Nous avons décrit l'ensemble de l'interface. Nous détaillons son fonctionnement à travers les méthodes utilisées et les changements de contexte. Les différents algorithmes ci-dessous ne tiennent pas compte de la mise en place des compatibilités au cours des différents appels (pour cela, on peut se reporter au chapitre 2 qui indique quand et à quelles conditions ces changements doivent être appliqués). Le premier cas étudié est celui du lancement d'une nouvelle action globale.




```
Menu.Activer ( )
   {Action_Globale est l'action associée au menu Menu}
   {Type_Menu est le type d'appel du menu : local, immédiat, différé}
   Action_Globale.Activer (Type_Menu)
```

```
Action_Globale.Activer (Type_Action)
   suivant Type_Action faire
      immédiat :
      { lancement de l'action et si elle n'est pas terminée, elle est mise au sommet de la
        pile des actions}
           courant = depart.activer ( )
           si courant ≠ VIDE    {l'action est en attente}
           alors Interface_Courante.Pile.Empiler(this, Type_Action)
      différé :
      { désactivation des actions de la pile et lancement de l'action avec sa nouvelle
        compatibilité}
           Act = Interface_Courante.Pile.Sommet ( ).Action ( )
           Act.Desactiver ( )
           Activer (Nouveau_Type_Action)
      local :
      { lancement de l'action et si elle n'est pas terminée, elle est mise au sommet de la
        pile des actions sinon on reactive l'action précédente avec le résultat }
           courant = depart.activer ( )
           si courant ≠ VIDE    {l'action est en attente}
           alors Interface_Courante.Pile.Empiler(this, Type_Action)
           sinon Act = Interface_Courante.Pile.Sommet ( ).Action ( )
                 Act.Reactiver (ActRes.Donner_Resultat ( ))
```

Le deuxième cas que nous considérons représente la réponse d'un opérateur à une interaction.




```
Fenetre.Identifier (Coordonnees)
   {Action_Globale est l'action associée au menu Menu}
   {Type_Menu est le type d'appel du menu : local, immédiat, différé}
   objet = résultat de l'identification en (x,y)
   Action_Globale.Reactiver (objet)
```

```
Action_Globale.Reactiver (objet)
   courant = courant.Reactiver (objet)
   si courant = VIDE   {l'action est terminée}
   alors Type_Action = Interface_Courante.Pile.Sommet ( ).Type ( )
         suivant Type_Action faire
            immédiat :
            { l'action qui vient de se terminer était immédiate : aucun résultat à
              répercuter, on dépile juste l'action pour reprendre la précédente}
              Interface_Courante.Pile.Depiler ( )
            local :
            { l'action qui vient de se terminer était locale : le résultat doit servir à
              réactiver l'action précédente, on dépile puis on réactive}
              Interface_Courante.Pile.Depiler ( )
              Act = Interface_Courante.Pile.Sommet ( ).Action ( )
              Act.Reactiver (ActRes.Donner_Resultat ( ))
            { ActRes est l'action contenant le résultat de l'action globale}
   sinon {l'action n'est pas terminée, on continue l'attente}
```

Pour la mise à jour des compatibilités, la gestion est très simple, il suffit que l'interface les modifie à chaque changement de contexte.

6. Exemple de l'implantation.

L'objectif d'implémenter un nouveau noyau est atteint. Nous l'avons défini de telle sorte qu'il soit indépendant de la boîte à outils utilisée. Ce noyau étant réalisé avec des widgets, nous avons voulu qu'il respecte leur philosophie. Le fait qu'une action globale interrompue rende la main à l'interface pour permettre d'y répondre permet de ne pas avoir de boucle d'attente d'événements et donc de laisser un maximum de contrôle aux éléments d'interface (widgets). Bien que récent, ce noyau a déjà servi de base à de nombreuses implantations.

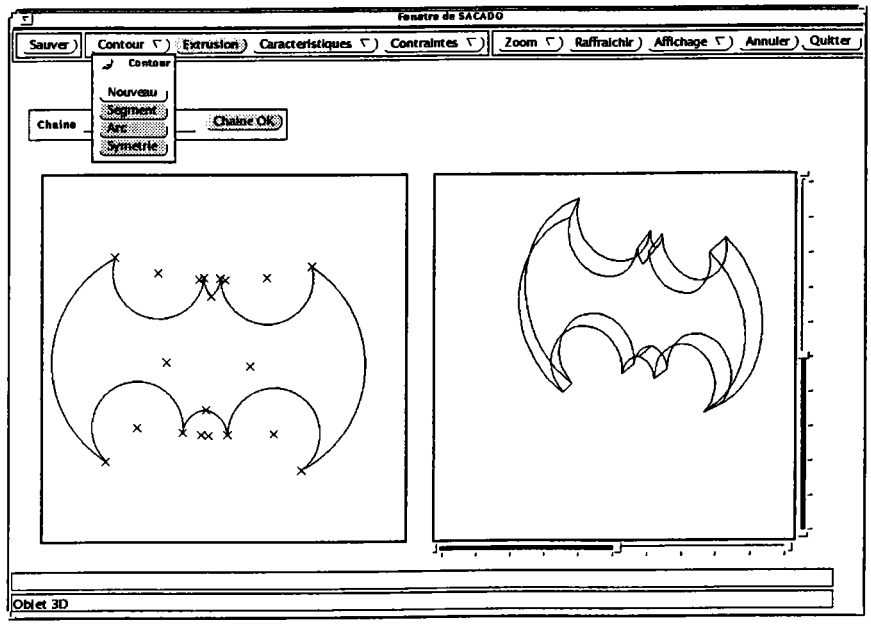


Figure A.8. *Une implantation SACADO.*

Une première utilisation a été faite dans le cadre d'une thèse d'ingénieur C.N.A.M. [PIP 95] mais l'outil présenté dans la figure A.8 a été réalisé comme implantation des travaux effectués lors de trois stages de D.E.A. [GAR 95a]. Il est opérationnel et s'appuie en particulier sur un dialogue dynamique avec détection des contraintes lors de la création des objets (interactions floues). Cette réalisation a permis de valider notre nouveau noyau et d'envisager de nouveaux développements dont l'ajout des bibliothèques que l'on peut trouver en annexe C.

ANNEXE B.

LE LANGAGE NADRAG.

1. INTRODUCTION.

NADRAG est un langage textuel dont le domaine privilégié est les applications graphiques interactives et plus particulièrement, la C.F.A.O. Il doit permettre la description complète de l'architecture fonctionnelle du système et sa syntaxe fait référence aussi bien à des actions interactives qu'à une description alphanumérique (compatibilités entre menus, types d'objets, ...). Le langage a été développé à l'aide de grammaires et du langage C++ en respectant la norme ANSI.

Nous présentons dans cette annexe une version implémentée de NADRAG qui diffère quelque peu de celle introduite dans le chapitre 3. C'est une version antérieure développée dans un but d'évaluation indépendamment de toute implantation SACADO. C'est la seule version existante à l'heure actuelle [MAR 92b]. Elle ne bénéficie pas des opérateurs de composition et les instructions du dialogue ne respectent pas le schéma décrit. Nous ne présentons que les grammaires du langage, pour la programmation proprement dite, on peut consulter [MAR 92].

2. LES GRAMMAIRES.

2.1. La grammaire lexicale.

Cette grammaire décrit l'ensemble des termes à reconnaître pour construire le langage (les objets dont le nom est précédé par ~ ne sont présents que pour des facilités d'écriture).

```
~alphabet          →  (' '..'&') | ('('..'~');
~chiffre           →  '0'..'9';
~entier            →  ~chiffre.(~chiffre)*;
~exp               →    ~lchif
                      | ~signe.~lchif;
~lchif             →  ~chiffre | ~chiffre.~chiffre;
~lettre            →  'A'..'Z';
~signe             →  '+' | '-';
"And"              →  'A'.'N'.'D';
"Begin"            →  'B'.'E'.'G'.'I'.'N';
"By"               →  'B'.'Y';
"Call"             →  'C'.'A'.'L'.'L';
"Case"             →  'C'.'A'.'S'.'E';
"Ccontrainte"      →  'C'.'O'.'N'.'T'.'R'.'A'.'I'.'N'.'T'.'E';
"Cdiffere"         →  'D'.'I'.'F'.'F'.'E'.'R'.'E';
"Cdomaine"         →  'D'.'O'.'M'.'A'.'I'.'N'.'E';
"Ceffet"           →  'E'.'F'.'F'.'E'.'T';
"Chaine"           →  ''''.(~alphabet | (''''.''''))*.'''';
"Cimmediat"        →  'I'.'M'.'M'.'E'.'D'.'I'.'A'.'T';
"Cinactif"         →  'I'.'N'.'A'.'C'.'T'.'I'.'F';
"Cinter"           →  'I'.'N'.'T'.'E'.'R'.'A'.'C'.'T'.'I'.'O'.'N';
"Clocal"           →  'L'.'O'.'C'.'A'.'L';
"Cmenu"            →  'M'.'E'.'N'.'U';
"Cmessage"         →  'M'.'E'.'S'.'S'.'A'.'G'.'E';
"Def"              →  'D'.'E'.'F';
"Div"              →  'D'.'I'.'V';
"Else"             →  'E'.'L'.'S'.'E';
"End"              →  'E'.'N'.'D';
"Endif"            →  'E'.'N'.'D'.'I'.'F';
"Endloop"          →  'E'.'N'.'D'.'L'.'O'.'O'.'P';
"Endswitch"        →  'E'.'N'.'D'.'S'.'W'.'I'.'T'.'C'.'H';
```

```
"Entier"          → ~entier;
"False"           → 'F'.'A'.'L'.'S'.'E';
"From"            → 'F'.'R'.'O'.'M';
"Identificateur"  → ~lettre.(~lettre | '_' | ~chiffre)*;
"If"              → 'I'.'F';
"Input"           → 'I'.'N'.'P'.'U'.'T';
"Loop"            → 'L'.'O'.'O'.'P';
"Mod"             → 'M'.'O'.'D';
"Not"             → 'N'.'O'.'T';
"On"              → 'O'.'N';
"Or"              → 'O'.'R';
"Otherwise"       → 'E'.'L'.'S'.'E'.'S'.'W'.'I'.'T'.'C'.'H';
"Output"          → 'O'.'U'.'T'.'P'.'U'.'T';
"Proc"            → 'P'.'R'.'O'.'C'.'E'.'D'.'U'.'R'.'E';
"Reel"            → ~entier.'.'.~entier
                    | ~entier.('e'|'E').~exp
                    | ~entier.'.'.~entier.('e'|'E').~exp;
"Result"          → 'R'.'E'.'S'.'U'.'L'.'T';
"Step"            → 'S'.'T'.'E'.'P';
"Switch"          → 'S'.'W'.'I'.'T'.'C'.'H';
"Then"            → 'T'.'H'.'E'.'N';
"Times"           → 'T'.'I'.'M'.'E'.'S';
"To"              → 'T'.'O';
"True"            → 'T'.'R'.'U'.'E';
"Until"           → 'U'.'N'.'T'.'I'.'L';
"While"           → 'W'.'H'.'I'.'L'.'E';
"Xor"             → 'X'.'O'.'R';
"+"               → '+';
"-"               → '-';
"*"               → '*';
"/"               → '/';
";"               → ';';
","               → ',';
"("               → '(';
")"               → ')';
":"               → ':';
":="              → ':'.'=';
"="               → '=';
"<>"              → '<'.'>';
"<"               → '<';
"<="              → '<'.'=';
">"               → '>';
">="              → '>'.'=';
"["               → '[';
"]"               → ']';
```

2.2. La Grammaire syntaxique.

Cette grammaire définit la syntaxe du langage, elle décrit l'ordonnacement des termes que l'on a introduits avec la grammaire lexicale. Le caractère & représente une phrase vide.

```
Programme          →   Resultat Liste_Modules Corps_Principal
Resultat           →   &
Resultat           →   'Resultat' Identificateur ';'
Identificateur     →   'Identificateur'
Liste_Modules      →   &
Liste_Modules      →   Liste_Modules Module
Module             →   'Proc' Identificateur ';' Liste_NADactions 'End' ';'
Corps_Principal    →   'Begin' Liste_NADactions 'End' ';'
Liste_NADactions   →   &
Liste_NADactions   →   Liste_NADactions NADaction
```

```
NADaction          →   Identificateur ';'
NADaction          →   Appel_Fonction ';'
Appel_Fonction     →   Identificateur '(' Liste_Expressions ')'
Appel_Fonction     →   Identificateur '(' ')'
Liste_Expressions  →   Expression
Liste_Expressions  →   Liste_Expressions ',' Expression
NADaction          →   'If' Expression 'Then' Liste_NADactions 'Endif' ';'
NADaction          →   'If' Expression 'Then' Liste_NADactions 'Else' Liste_NADactions
                       'Endif' ';'
NADaction          →   'Def' Identificateur 'By' Expression ';'
NADaction          →   'Def' Acces_Tableau 'By' Expression ';'
Acces_Tableau      →   Identificateur '[' Liste_Expressions ']'
NADaction          →   Identificateur ':=' Expression ';'
NADaction          →   Acces_Tableau ':=' Expression ';'
NADaction          →   'Switch' Liste_Cas 'Endswitch' ';'
NADaction          →   'Switch' Liste_Cas Else_Switch 'Endswitch' ';'
NADaction          →   'Switch' Expression Liste_Cas 'Endswitch' ';'
NADaction          →   'Switch' Expression Liste_Cas Else_Switch 'Endswitch' ';'
Liste_Cas          →   Cas
Liste_Cas          →   Liste_Cas Cas
Cas                →   'Case' Expression ':' Liste_NADactions
Else_Switch        →   'Otherwise' ':' Liste_NADactions
Expression         →   Expression 'And' Sous_Expr_Comp
Expression         →   Expression 'Or' Sous_Expr_Comp
Expression         →   Expression 'Xor' Sous_Expr_Comp
Expression         →   Sous_Expr_Comp
Sous_Expr_Comp     →   Sous_Expr_Ajout '=' Sous_Expr_Ajout
Sous_Expr_Comp     →   Sous_Expr_Ajout '<>' Sous_Expr_Ajout
Sous_Expr_Comp     →   Sous_Expr_Ajout '<' Sous_Expr_Ajout
Sous_Expr_Comp     →   Sous_Expr_Ajout '<=' Sous_Expr_Ajout
Sous_Expr_Comp     →   Sous_Expr_Ajout '>' Sous_Expr_Ajout
Sous_Expr_Comp     →   Sous_Expr_Ajout '>=' Sous_Expr_Ajout
Sous_Expr_Comp     →   Sous_Expr_Ajout
Sous_Expr_Ajout    →   Sous_Expr_Ajout '+' Sous_Expr_Mult
Sous_Expr_Ajout    →   Sous_Expr_Ajout '-' Sous_Expr_Mult
Sous_Expr_Ajout    →   Sous_Expr_Mult
Sous_Expr_Mult     →   Sous_Expr_Mult '*' Terme
Sous_Expr_Mult     →   Sous_Expr_Mult '/' Terme
Sous_Expr_Mult     →   Sous_Expr_Mult 'Div' Terme
Sous_Expr_Mult     →   Sous_Expr_Mult 'Mod' Terme
Sous_Expr_Mult     →   Terme
Terme              →   '(' Expression ')'
Terme              →   'Not' Terme
Terme              →   '+' Terme
Terme              →   '-' Terme
Terme              →   'Entier'
Terme              →   'Reel'
Terme              →   'True'
Terme              →   'False'
Terme              →   'Chaine'
Terme              →   Identificateur
Terme              →   Acces_Tableau
Terme              →   Appel_Fonction
Terme              →   'Ccontrainte' '(' Expression ')'
Terme              →   'Ceffet' '(' Liste_NADactions ')'
Terme              →   'Clocal' '(' Nom_Elt ')'
Terme              →   'Cdiffere' '(' Nom_Elt ')'
Terme              →   'Cimmediat' '(' Nom_Elt ')'
Terme              →   'Cinactif' '(' Nom_Elt ')'
Terme              →   'Cinter' '(' Identificateur ')'
Terme              →   'Cmessage' '(' 'Chaine' ')'
Nom_Elt            →   'Chaine'
```

```
Nom_Elt          →    'Cdomaine' '(' 'Chaine' ')'
Nom_Elt          →    'Cmenu' '(' 'Chaine' ')'
```

3. EXEMPLE.

L'exemple que nous avons choisi est celui d'une action globale de création d'un segment par deux points différents. Chaque objet du dialogue est créé avant son utilisation (instruction DEF ... BY) et le déclenchement d'une interaction se fait par simple affectation (objet := interaction).

```
{ indication du résultat de l'action globale : le cercle construit }
RESULTAT cercle;
{ construction de l'interaction pour le centre }
DEF InterCentre    BY INTERACTION ( Point );
DEF InterCentre    BY InterCentre + MESSAGE ( "Donner le centre du cercle" );
DEF InterCentre    BY InterCentre + LOCAL ( MENU ( "Point" ));
{saisie du centre}
centre := InterCentre;
{construction de l'interaction pour le rayon}
DEF InterRayon     BY INTERACTION ( Scalaire );
DEF InterRayon     BY InterRayon + CONTRAINTE ( InterRayon > 0 );
DEF InterRayon     BY InterRayon + MESSAGE ( "Donner le rayon du cercle" );
DEF InterRayon     BY InterRayon + LOCAL ( MENU ( "Distance" ));
{ saisie du rayon }
rayon := InterRayon ;
{création du cercle}
cercle := creer_cercle (centre, rayon);
```

Cet exemple montre une simplification importante par rapport à la description initiale qu'il faut faire avec le noyau décrit dans l'annexe A. La compréhension s'en trouve facilitée et le nombre d'erreurs est considérablement réduit car l'interpréteur indique les erreurs alors que l'instanciation effectuée avec le noyau ne permettait pas de vérification.

4. CONCLUSION.

Même si cette version de NADRAG ne correspond pas aux derniers travaux effectués, ce langage a montré que l'on pouvait décrire entièrement l'architecture. Un travail important reste à accomplir pour aboutir à un langage complet et intégré mais nos travaux se sont concentrés vers la spécification du langage graphique et vers la génération du dialogue. À terme, notre objectif est de développer une nouvelle version prenant en compte toutes les instructions du dialogue et de l'intégrer totalement dans la version actuelle de SACADO.

ANNEXE C.

LES BIBLIOTHÈQUES DE SACADO.

1. INTRODUCTION.

Les travaux réalisés sur les bibliothèques d'interactions et sur le formalisme graphique nous ont amené à modifier l'implémentation de SACADO. En effet, la disponibilité d'outils de développement nous impose la possibilité de paramétrer le noyau de développement de manière simple. Dans la version actuelle, le composant présentation (domaines, menus, ...) et les actions globales sont définis de manière statique en utilisant le langage C++, langage de développement de SACADO. Toute modification entraîne une recompilation de l'application.

Afin de faciliter le paramétrage du noyau par des outils externes, nous avons décidé d'ajouter trois bibliothèques qui représentent un ensemble de ressources :

- une bibliothèque de présentation. Elle contient toute la description de la partie statique de l'interface avec laquelle l'opérateur interagit (c'est l'aspect externe). Cette bibliothèque est représentée par un fichier unique;
- une bibliothèque d'actions globales. Elle contient l'ensemble des actions globales à considérer pour une implantation. En fait, cette bibliothèque est représentée par un ensemble de fichiers, chaque fichier correspondant à une action globale;
- une bibliothèque d'interactions. Elle contient pour chaque interaction, les compatibilités à utiliser. Cette description est faite dans un fichier unique.

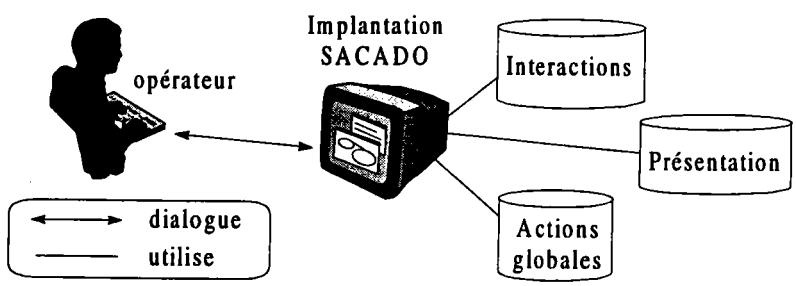


Figure C.1. *Les bibliothèques de SACADO.*

Toutes ces bibliothèques sont décrites sous forme de fichiers textes. Nous avons donc utilisé des interpréteurs pour analyser ces fichiers. Dans les paragraphes qui suivent, nous donnons les grammaires qui correspondent à chaque bibliothèque. Un exemple est utilisé pour en illustrer la syntaxe.

2. LE COMPOSANT PRÉSENTATION.

Cette bibliothèque répond au besoin de décrire l'aspect externe de l'interface SACADO. On y trouve notamment les domaines, les menus mais également les fenêtres d'application, les fenêtres de travail (zones d'interaction avec l'opérateur), zones d'entrée des données, ... Pour chacun de ces objets, on trouve une instruction pour le décrire. La bibliothèque est représentée par un seul fichier de description qui est lu lors de l'initialisation.

2.1. Les grammaires.

2.1.1. La grammaire lexicale.

```
~alphabet          →  (' '..'&') | ('('..'~');
~chiffre           →  '0'..'9';
~entier            →  ~chiffre.(~chiffre)*;
~signe             →  '+' | '-';
~lchif             →  ~chiffre | ~chiffre.~chiffre;
~exp               →      ~lchif
                   |   ~signe.~lchif;
"Affichage"        →  'A'.'F'.'F'.'I'.'C'.'H'.'A'.'G'.'E';
"Chaine"           →  '"'.(~alphabet)*.'"';
"Compatibilite"    →  'C'.'O'.'M'.'P'.'A'.'T'.'I'.'B'.'I'.'L'.'I'.'T'.'E';
"Complete"         →  'C'.'O'.'M'.'P'.'L'.'E'.'T'.'E';
"De"               →  'D'.'E';
"Differe"          →  'D'.'I'.'F'.'F'.'E'.'R'.'E'    ;
"Domaine"          →  'D'.'O'.'M'.'A'.'I'.'N'.'E'    ;
"Est"              →  'E'.'S'.'T';
"Fenetre"          →  'F'.'E'.'N'.'E'.'T'.'R'.'E';
"Fils"             →  'F'.'I'.'L'.'S';
"Groupe"           →  'G'.'R'.'O'.'U'.'P'.'E';
"Icone"            →  'I'.'C'.'O'.'N'.'E';
"Immediat"         →  'I'.'M'.'M'.'E'.'D'.'I'.'A'.'T';
"Inactif"          →  'I'.'N'.'A'.'C'.'T'.'I'.'F';
"Interface"        →  'I'.'N'.'T'.'E'.'R'.'F'.'A'.'C'.'E';
"Local"            →  'L'.'O'.'C'.'A'.'L';
"Menu"             →  'M'.'E'.'N'.'U';
"Normal"           →  'N'.'O'.'R'.'M'.'A'.'L';
"Numero"           →  ~entier
"Option"           →  'O'.'P'.'T'.'I'.'O'.'N';
"Reel"             →      ~entier.'.'.~entier
                   |   ~entier.('e'|'E').~exp
                   |   ~entier.'.'.~entier.('e'|'E').~exp;
"Saisie"           →  'S'.'A'.'I'.'S'.'I'.'E';
"Scalaire"         →  'S'.'C'.'A'.'L'.'A'.'I'.'R'.'E';
"Texte"            →  'T'.'E'.'X'.'T'.'E'     ;
"Vide"             →  'V'.'I'.'D'.'E';
"2d"               →  '2'.'D';
":"                →  ':';
";"                →  ';';
"["                →  ';';
"]"                →  ';';
","                →  ',';
```

2.1.2. La grammaire syntaxique.

```
Application        → EnTete SuActions SuCompatibilites
EnTete             → 'Interface' 'Complete' ':'
SuActions          → ListeActions
ListeActions       → &
ListeActions       → Actions ';' ListeActions
Actions            → Domaine
Actions            → Interface
Actions            → GroupeSans
Actions            → GroupeAvec
Actions            → Terminal
Actions            → Fenetre2D
Actions            → Texte
Actions            → Scalaire
```

```
Domaine          → 'Domaine' ':' Element '[' Liste ']' '[' Defaut ']'
Interface        → 'Fenetre' 'Interface' ':' NomFenetre Element LargMin LargMax HautMin
                   HautMax Ouverture X Y Largeur Hauteur 'Fils' 'De' Parent '[' Liste ']'
                   '[' Defaut ']'
GroupeSans       → 'Groupe' ':' Element 'Fils' 'De' Parent '[' Liste ']' '[' Defaut ']'
GroupeAvec       → 'Groupe' ':' Element X Y 'Fils' 'De' Parent '[' Liste ']' '['
                   Defaut']'
Terminal         → 'Option' 'Menu' ':' Element 'Fils' 'De' Parent '[' Liste ']' '['
                   Defaut ']'
Fenetre2D        → 'Fenetre' '2D' ':' Element Scene XFen YFen LargFen HautFen X Y Largeur
                   Hauteur 'Fils' 'De' Parent '[' Liste ']' '[' Defaut ']'
Texte            → 'Affichage' 'Texte' ':' Element X Y Largeur Hauteur 'Fils' 'De' Parent
                   '[' Liste ']' '[' Defaut ']'
Texte            → 'Saisie' 'Texte' ':' Element 'Fils' 'De' Parent '[' Liste ']' '['
                   Defaut ']'
Scalaire         → 'Saisie' 'Scalaire' ':' Element 'Fils' 'De' Parent '[' Liste ']' '['
                   Defaut ']'
Parent           → 'Chaine'
Element          → 'Chaine'
NomFenetre       → 'Chaine'
LargMin          → 'Numero'
LargMax          → 'Numero'
HautMin          → 'Numero'
HautMax          → 'Numero'
Ouverture        → 'Normal'
Ouverture        → 'Icone'
X                → 'Numero'
Y                → 'Numero'
Largeur          → 'Numero'
Hauteur          → 'Numero'
Scene            → 'Numero'
XFen             → 'Reel'
YFen             → 'Reel'
LargFen          → 'Reel'
HautFen          → 'Reel'
Liste            → 'Vide'
Liste            → ActGlob ',' ListeNonVide
Liste            → ActGlob
ActGlob          → 'Chaine'
ListeNonVide     → ActGlob ',' ListeNonVide
ListeNonVide     → ActGlob
Defaut           → 'Vide'
Defaut           → DefCompat
SuCompatibilites → &
SuCompatibilites → Compatibilites ';' SuCompatibilites
Compatibilites   → 'Compatibilite' ':' Element 'Est' DefCompat 'De' Element
DefCompat        → 'Immediat'
DefCompat        → 'Local'
DefCompat        → 'Differe'
DefCompat        → 'Inactif'
```

2.2. Exemple.

Pour illustrer la syntaxe du composant présentation, nous avons choisi de présenter le fichier qui correspond à l'exemple de la figure II.10. On obtient la ressource suivante :

```
INTERFACE COMPLETE :

DOMAINE : "Domaine Principal" ["Initialisation"][Immediat];

FENETRE INTERFACE : "Fenetre principale SACADO" "Sacado" 100 1000 100 800
NORMAL 300 250 600 580 FILS DE "Domaine Principal" [VIDE][VIDE];
```

```
GROUPE : "Groupe" FILS DE "Fenetre principale SACADO" [VIDE][VIDE];

OPTION MENU : "Quitter" FILS DE "Groupe de Menus" ["Annuler"][VIDE];
OPTION MENU : "Segment" FILS DE "Groupe de Menus" ["S2Pts"][VIDE];
OPTION MENU : "Cercle" FILS DE "Groupe de Menus" ["CrayET"][VIDE];
OPTION MENU : "Point" FILS DE "Groupe de Menus" [VIDE][VIDE];
OPTION MENU : "Coordonnees" FILS DE "Point" ["Coordonnees"][VIDE];
OPTION MENU : "Intersection" FILS DE "Point" ["Intersection"][VIDE];

FENETRE 2D : "Fenetre 2D" 1 -1000.0 -1000.0 2000.0 2000.0 10 50 500 500 FILS
DE "Fenetre principale SACADO" ["Raffraichir2D","Identifier2D"][Immediat];

AFFICHAGE TEXTE : "Message" 0 -1 -1 20 FILS DE "Fenetre principale SACADO"
[VIDE][VIDE];

COMPATIBILITE : "Segment" EST DIFFERE DE "Cercle";
```

Cette ressource construit une interface basée sur un domaine unique, le domaine principal (c'est le domaine propre). Ce domaine est constitué d'une fenêtre d'application ("Fenetre principale SACADO"), d'un groupe de menus ("Groupe"), d'une fenêtre de travail ("Fenetre 2D") et d'une zone d'affichage de texte ("Message"). Chaque élément est paramétré en fonction de son type. On obtient finalement l'interface suivante :

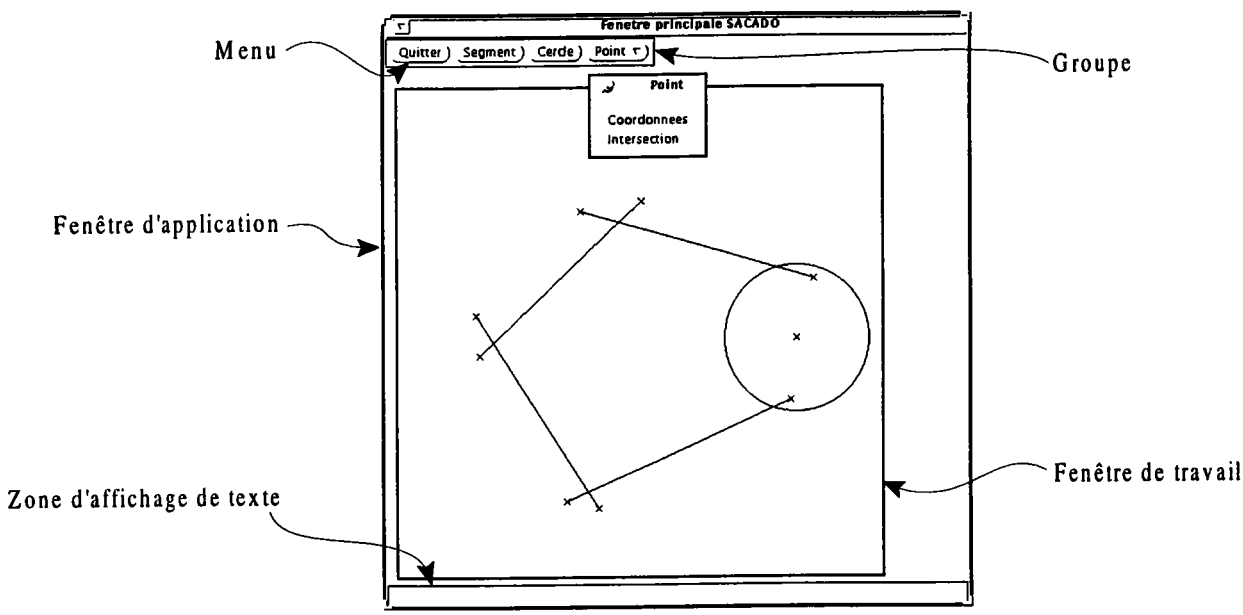


Figure C.2. *Exemple d'interface SACADO.*

Pour chaque élément, on doit indiquer entre crochets les actions globales associées et la compatibilité de départ. Par exemple pour le domaine principal, la compatibilité indique que, par héritage, toute l'interface est immédiate (lorsque VIDE apparaît cela signifie qu'aucune compatibilité particulière n'est requise). On peut préciser les compatibilités entre les différents éléments grâce à l'instruction COMPATIBILITE. Dans l'exemple, le menu SEGMENT a été déclaré différé du menu CERCLE. Concernant les actions globales, il s'agit de préciser les actions ayant un sens pour l'élément. Un domaine comporte une seule action globale (action d'initialisation) alors qu'une fenêtre de travail en comporte deux (une action de reaffichage du contenu et une action d'identification des objets). Pour les menus, il faut indiquer l'action globale qui réalise l'opération désirée. Par exemple, c'est l'action globale

"S2Pts" qui décrit la création d'un segment à partir de deux points. Nous décrivons dans le paragraphe suivant, le format que nous avons retenu pour la description de ces actions globales.

3. LES ACTIONS GLOBALES.

Nous avons décidé de représenter la bibliothèque des actions globales par un ensemble de fichiers, chaque fichier décrivant une action globale. Par exemple, pour l'action globale "S2Pts", qui est associée au menu SEGMENT, sa description doit se trouver dans le fichier dont le nom est "S2Pts.ag" (ag pour action globale). Les deux grammaires que nous donnons s'appliquent donc à la description d'une seule action globale et respectent totalement les entités introduites lors de la description du formalisme graphique. La seule contrainte provient du noyau de développement de SACADO qui nécessite des actions globales déterministes (comme nous le verrons dans l'exemple, à charge de l'outil de description de rendre déterministes les actions non déterministes).

3.1. Les grammaires.

3.1.1. La grammaire lexicale.

```
~alphabet          →  (' '..'&') | ('('..'~');
~lettre            →  'A'..'Z';
~chiffre           →  '0'..'9';
~entier            →  ~chiffre.(~chiffre)*;
"Ani"              →  'A'.'N'.'I';
"Aniar"            →  'A'.'N'.'I'.'A'.'R'     ;
"Bouclage"         →  'B'.'O'.'U'.'C'.'L'.'A'.'G'.'E';
"Chaine"           →  '"'.(~alphabet)*.'"';
"Code"             →  'C'.'O'.'D'.'E';
"Contrainte"       →  'C'.'O'.'N'.'T'.'R'.'A'.'I'.'N'.'T'.'E';
"De"               →  'D'.'E';
"Depart"           →  'D'.'E'.'P'.'A'.'R'.'T';
"Effet"            →  'E'.'F'.'F'.'E'.'T';
"Et"               →  'E'.'T';
"Faux"             →  'F'.'A'.'U'.'X';
"FlotDeDonnee"     →  'F'.'L'.'O'.'T'.' '.'D'.'E'.' '.'D'.'O'.'N'.'N'.'E'.'E';
"Interaction"      →  'I'.'N'.'T'.'E'.'R'.'A'.'C'.'T'.'I'.'O'.'N';
"Nom"              →  ~lettre.(~lettre | '_' | ~chiffre)*;
"Nombre"           →  'N'.'O'.'M'.'B'.'R'.'E';
"Numero"           →  ~entier
                      | '-'.~entier;
"Ou"               →  'O'.'U';
"Parametre"        →  'P'.'A'.'R'.'A'.'M'.'E'.'T'.'R'.'E';
"Parametres"       →  'P'.'A'.'R'.'A'.'M'.'E'.'T'.'R'.'E'.'S';
"Position"         →  'P'.'O'.'S'.'I'.'T'.'I'.'O'.'N';
"Resultat"         →  'R'.'E'.'S'.'U'.'L'.'T'.'A'.'T';
"Sequence"         →  'S'.'E'.'Q'.'U'.'E'.'N'.'C'.'E';
"Type"             →  'T'.'Y'.'P'.'E';
"Vers"             →  'V'.'E'.'R'.'S';
"Vrai"             →  'V'.'R'.'A'.'I';
":"                →  ':';
":"                →  ':';
","                →  ';';
"("                →  '(';
")"                →  ')';
```

3.1.2. La grammaire syntaxique.

```
ActionsGlobales   → Parametres SuParametres ListeDefinitions Divers
Parametres        → 'Nombre' 'De' 'Parametres' ':' 'Numero' ';'
SuParametres      → &
SuParametres      → Parametres ';' SuParametres
Parametres        → 'Parametre' ':' Nom '(' 'Numero' ')' 'Position' ':' 'Numero'
Nom               → &
Nom               → 'Nom'
ListeDefinitions  → &
ListeDefinitions  → Definitions ';' ListeDefinitions
Definitions       → Interactions
Definitions       → OperateursET
Definitions       → OperateursOU
Definitions       → Ani
Definitions       → Aniar
Definitions       → Contraintes
Definitions       → Effets
Definitions       → Sequences
Definitions       → FlotsDeDonnees
Interactions      → 'Interaction' ':' Nom '(' 'Numero' ')' 'Chaine' 'Type' ':' 'Nom'
Interactions      → 'Interaction' ':' Nom '(' 'Numero' ')' 'Chaine' 'Type' ':' 'Numero'
OperateursET      → 'Et' Nom '(' 'Numero' ')' ':' PremiereOperande ',' ListeNom
OperateursOU      → 'Ou' Nom '(' 'Numero' ')' ':' PremiereOperande ',' ListeNom
PremiereOperande  → Nom '(' 'Numero' ')'
ListeNom          → Nom '(' 'Numero' ')'
ListeNom          → ListeNom ',' Nom '(' 'Numero' ')'
Aniar             → 'Aniar' ':' Nom '(' 'Numero' ')' 'Chaine' 'Code' ':' 'Numero'
                    'Parametre' ':' 'Numero'
Ani               → 'Ani' ':' Nom '(' 'Numero' ')' 'Chaine' 'Code' ':' 'Numero'
                    'Parametre' ':' 'Numero'
Contraintes       → 'Contrainte' ':' Nom '(' 'Numero ')' 'Chaine' 'Code' ':' 'Numero'
                    'Parametre' ':' 'Numero' 'Vers' Nom '(' 'Numero' ')'
Contraintes       → 'Contrainte' ':' Nom '(' 'Numero' ')' 'Code' ':' 'Numero' 'Parametre'
                    ':' 'Numero' 'Vers' Nom '(' 'Numero' ')'
Effets            → 'Effet' ':' Nom '(' 'Numero' ')' 'Chaine' 'Code' ':' 'Numero'
                    'Parametre' ':' 'Numero' 'Type' ':' 'Nom' 'Vers' Nom '(' 'Numero' ')'
Sequences         → 'Sequence' ':' Nom '(' 'Numero' ')' 'Vers' Nom '(' 'Numero' ')'
FlotsDeDonnees    → 'FlotDeDonnee' ':' Nom '(' 'Numero' ')' 'Vers' Nom '(' 'Numero' ')'
FlotsDeDonnees    → 'FlotDeDonnee' ':' Nom '(' 'Numero' ')' 'Vers' Nom '(' 'Numero' ')'
Divers            → Depart Resultat Bouclage
Depart            → 'Depart' ':' Nom '(' 'Numero' ')' ';'
Resultat          → 'Resultat' ':' Nom '(' 'Numero' ')' ';'
Resultat          → &
Bouclage          → 'Bouclage' ':' 'Vrai' ';'
Bouclage          → 'Bouclage' ':' 'Faux' ';'
```

3.2. Exemple.

L'exemple que nous avons choisi s'inscrit dans la suite de l'exemple déjà proposé. C'est l'action globale de création d'un segment à partir de deux points distincts. Grâce au formalisme graphique, on obtient pour cette action le schéma non déterministe suivant :

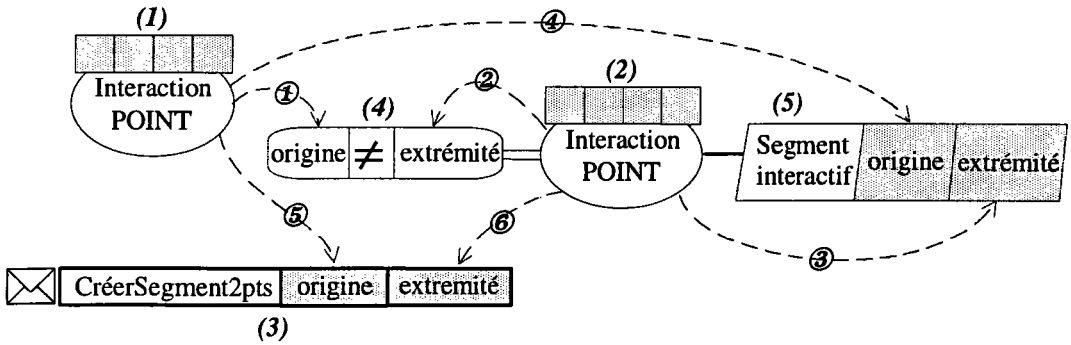


Figure C.3. *Action globale de création d'un segment à partir de deux points distincts.*

Comme nous l'avons précisé, le noyau SACADO s'appuie sur la description déterministe des actions globales. Il convient donc à partir de la description précédente de déduire la séquence des différentes actions. C'est pourquoi, dans le fichier qui suit, la séquence apparaît entre les interactions et les actions non interactives. Elle est déduite du schéma et on trouve les séquences (1) → (2) et (4) → (3).

```
NOMBRE DE PARAMETRES : 0;

INTERACTION: (1)  "Premier point"  TYPE : TypePoints;
INTERACTION: (2)  "Second point"  TYPE : TypePoints;

ANIAR: (3)  "Segment 2 points"  CODE : 1  PARAMETRE : 2;

CONTRAINTE: (4)  "Difference"  CODE : 9  PARAMETRE : 2  VERS (2);

COMPORTEMENT: (5)  "Segment elastique"  CODE : 1  PARAMETRE : 2  VERS (2);

SEQUENCE: (1)  VERS (2);
SEQUENCE: (2)  VERS (3);

FLOT DE DONNEE: (1)  VERS (4); ①
FLOT DE DONNEE: (2)  VERS (4); ②
FLOT DE DONNEE: (2)  VERS (5); ③
FLOT DE DONNEE: (1)  VERS (5); ④
FLOT DE DONNEE: (1)  VERS (3); ⑤
FLOT DE DONNEE: (2)  VERS (3); ⑥

DEPART    : (1);
RESULTAT : (3);
```

Dans ce fichier, on retrouve toutes les entités de la figure C.3. Cela devrait faciliter l'écriture d'un outil de génération des actions globales à partir de leur description graphique. Pour aider à la compréhension, des annotations que l'on retrouve sur le schéma et dans le fichier ont été ajoutées (par exemple *(1)* et ③). Le fichier de description est lu à chaque sélection d'un menu, cela autorise des évolutions dynamiques ("à la volée") d'une implantation. Si l'opérateur désire modifier une action, la modification est prise en compte dès l'exécution suivante. Cela conforte notre approche de vouloir permettre la modification de l'implantation pendant son exécution, de modifier de manière itérative un prototype opérationnel sans aucune phase de compilation qui serait pénalisante.

4. LES COMPATIBILITÉS.

La description des actions globales qui a été faite dans le paragraphe précédent ne comporte aucune mention aux compatibilités. En effet, elles sont définies dans une bibliothèque qui est valable pour toute l'implantation SACADO. De cette façon, le programmeur d'interfaces se concentre sur les actions ou sur les compatibilités mais sans se perdre dans les deux descriptions. Elles sont faites indépendamment l'une de l'autre. Comme dans le cas du composant présentation, cette bibliothèque est décrite par un fichier unique qui contient pour chaque type d'interaction, ses compatibilités en faisant référence aux objets définis dans le composant présentation de la même implantation.

4.1. Les grammaires.

4.1.1. La grammaire lexicale.

```
~alphabet          →  (' '..'&') | ('('..'~');
~lettre            →  'A'..'Z';
"Chaine"           →  '"'.(~alphabet)*.'"';
"Compatibilite"    →  'C'.'O'.'M'.'P'.'A'.'T'.'I'.'B'.'I'.'L'.'I'.'T'.'E';
"De"               →  'D'.'E';
"Differe"          →  'D'.'I'.'F'.'F'.'E'.'R'.'E';
"Est"              →  'E'.'S'.'T';
"Fin"              →  'F'.'I'.'N';
"Immediat"         →  'I'.'M'.'M'.'E'.'D'.'I'.'A'.'T';
"Interaction"      →  'I'.'N'.'T'.'E'.'R'.'A'.'C'.'T'.'I'.'O'.'N';
"Interface"        →  'I'.'N'.'T'.'E'.'R'.'F'.'A'.'C'.'E';
"Inactif"          →  'I'.'N'.'A'.'C'.'T'.'I'.'F';
"Local"            →  'L'.'O'.'C'.'A'.'L';
"Nom"              →  ~lettre.(~lettre | '_' | ~chiffre)*;
":"                →  ':';
";"                →  ';';
```

4.1.2. La grammaire syntaxique.

```
PartieCompatibilites          → 'Compatibilité' 'Interaction' ':' SuCompatibilites 'Fin'
SuCompatibilites              → &
SuCompatibilites              → CompatibilitesInteractions ';' SuCompatibilites

CompatibilitesInteractions    → 'Compatibilité' ':' 'Interface' 'Chaine' 'Est' DefCompat
                                'De' 'Interaction' 'Nom'
CompatibilitesInteractions    → 'Compatibilité' ':' 'Interface' 'Chaine' 'Est' DefCompat
                                'De' 'Interaction' 'Numero'
DefCompat                     → 'Immediat'
DefCompat                     → 'Local'
DefCompat                     → 'Differe'
DefCompat                     → 'Inactif'
```

4.2. Exemple.

Nous reprenons notre exemple du segment, les deux interactions *(1)* et *(2)* demandent un point. Le programmeur d'interfaces peut indiquer dans la bibliothèque les différentes compatibilités qu'il envisage pour une interaction de ce type. On obtient par exemple le fichier suivant :

```
COMPATIBILITE INTERACTION :

COMPATIBILITE : INTERFACE "Point" EST Local DE INTERACTION TypePoints ;

FIN
```

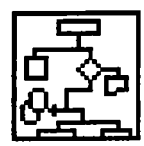
Le programmeur d'interfaces a simplement autoriser le menu POINT comme local à toutes les interactions désirant recevoir un point (`TypePoints`). Bien entendu d'autres compatibilités peuvent être définies mais pour notre exemple, c'est suffisant. Cette séparation entre les actions globales et les compatibilités autorise une ergonomie centralisée, toutes les interactions de même type possèdent les mêmes compatibilités. L'opérateur est alors confronté à un environnement qui offre toujours les mêmes possibilités, ce qui facilite l'apprentissage. De plus, une fois les compatibilités définies, les nouvelles actions globales en héritent naturellement ce qui soulage le travail à effectuer. C'est très important notamment lors du suivi des applications.

5. CONCLUSION.

Les trois bibliothèques qui ont été introduites dans la nouvelle implémentation de SACADO ouvrent la voie aux différents outils de génération du dialogue que nous envisageons. Ces outils peuvent modifier ou enrichir une implantation sans recourir à une compilation ou tout autre mécanisme coûteux en termes de temps. En particulier, ces bibliothèques permettent une intégration totale de l'outil basé sur le formalisme graphique. Ces premiers travaux réalisés en commun dans le cadre d'un stage de D.E.A. ont donné lieu à une première maquette de cet outil de manipulation des actions globales [LAH 95]. Cette maquette génère des fichiers dont la syntaxe correspond aux grammaires introduites précédemment. Cela autorise en particulier, un développement itératif d'une implantation alors même que le système est opérationnel (raffinements successifs). Cependant, cet outil n'a pas été développé avec SACADO, c'est un développement annexe et son intégration comme "menu" n'est pas aisée. Il est pourtant essentiel de pouvoir l'utiliser à n'importe quel moment de l'exécution du système. C'est pourquoi, nous nous sommes intéressés à son développement comme implantation SACADO. Il suffira alors de réunir cet outil et toute implantation SACADO pour obtenir un système de C.F.A.O. entièrement paramétrable. Pour réaliser ce nouveau prototype, il nous a semblé intéressant d'utiliser nos propositions sur la génération du dialogue à partir des comportements. C'est pourquoi, dans l'annexe D, nous montrons comment, à partir de la description des entités du formalisme par les comportements, nous avons généré un outil de manipulation.

ANNEXE D.

UN MODÈLE PAR LES COMPORTEMENTS.



1. INTRODUCTION.

Ce qui nous importe dans ce travail d'implémentation, ce n'est pas le fait de créer une application fonctionnelle mais la façon de la décrire et en particulier, la conception du dialogue. En réalité, l'application aurait pu porter sur n'importe quel domaine (réalisation de pièces paramétrées, description de circuits électroniques ...). Elle doit simplement permettre de valider nos théories en matière de génération de dialogue indépendamment de son usage. Le seul critère à respecter est de concevoir une application graphique interactive.

2. LE MODÈLE.

Nous présentons les différentes classes qui entrent en jeu dans la définition de notre modèle.

2.1. Les objets.

La définition des objets passe par trois classes distinctes. La première classe décrit un objet en termes de comportements (elle correspond à la spécification d'un objet).

```
class ObjetsSpecifies
{
            Producteurs*    TabProducteurs [NbMaxProducteurs];
            RIntrinseques*  TabRIntrinseques [NbMaxRIntrinseques];
            RExtrinseques*  TabRExtrinseques [NbMaxRExtrinseques];
            Transmuteurs*   TabTransmuteurs [NbMaxTransmuteurs];
            int             NumType;
public:
            méthodes ...
};
```

Il s'agit de décrire pour cet objet son type (NumType) et l'ensemble des comportements qui s'y rattache (TabProducteurs, TabRIntrinseques, TabRExtrinseques et TabTransmuteurs).

Les deux autres classes sont destinées à gérer les objets créés par l'opérateur lors de l'utilisation de l'application. On peut considérer que ce sont des instance des objets spécifiés précédemment. La classe *ObjetsApplications* représente l'objet en tant que tel. En fait, c'est la représentation canonique.

```
class ObjetsApplications
{
            ObjetsSpecifies*  OS;
            char*             NomType;
            int               NumObj;
            int               NumRExtrinsequeCourant;
public:
            méthodes ...
};
```

Un objet est déterminé par son type (NomType), par son identifiant qui est unique (NumObj) et par le réacteur extrinsèque utilisé pour son affichage (NumRExtrinsequeCourant). Le réacteur est représenté par un code, code utilisé pour le retrouver dans l'objet spécifié correspondant (OS). L'ensemble des objets d'application est représenté par la classe *ModelesObjetsApplications*.

```
class ModelesObjetsApplications
{
        ObjetsApplications*  TabObjets [2][NbMaxObjets];
        int                  NbObjets;
public:
        méthodes ...
};
```

Ce modèle contient tous les objets créés par l'opérateur lors de l'utilisation de l'application (TabObjets). Le nombre d'objets (NbObjet) permet de donner des identifiants uniques.

2.2. Les propriétés.

Une propriété est décrite avant tout pas sa sémantique représentée par la classe *Proprietes*.

```
class Proprietes
{
        char*           Nom;
        int             CodePropriete;
        int             Arite;
        int*            TabCodeRIDetection;
public:
        méthodes ...
};
```

Une propriété est définie par son nom (Nom), par un code unique de référence (CodePropriete), par son arité (Arite) et par les codes des réacteurs intrinsèques associés qui calculent les données (TabCodeRIDetection). L'ensemble des propriétés est représenté par la classe *ModelesProprietes*.

```
class ModelesProprietes
{
        Proprietes*     TProp [NbMaxProprietes];
        int             NbRIntrinseques;
public:
        méthodes ...
};
```

Ce modèle contient toutes les propriétés spécifiées par l'opérateur (TProp). Le nombre de réacteurs intrinsèques (NbRIntrinseques) permet de donner des identifiants uniques aux réacteurs intrinsèques.

2.3. Les réacteurs.

2.3.1. Les réacteurs extrinsèques.

Dans l'implémentation que nous considérons, les réacteurs extrinsèques ont été réduits à une simple primitive d'affichage. Tout réacteur est donc représenté par un code qui permet de faire référence à une primitive de l'implantation SACADO (CodeReacteur). Par exemple, le tracé d'un segment, ... Ce code fait également office d'identifiant pour le réacteur.

```
class RExtrinseques
{
        int             CodeReacteur;
        char*           Nom;
public:
        méthodes ...
};
```

2.3.2. Les réacteurs intrinsèques.

Un réacteur intrinsèque est défini par un identifiant unique qui permet d'y faire référence (CodeReacteur). Ce code correspond au code contenu dans la propriété correspondante. Il s'agit de créer le lien entre la propriété et l'un de ses réacteurs intrinsèques. Il est également nécessaire de préciser la fonction d'évaluation de la donnée qui est associée à ce réacteur (FonctionParametre) et on peut préciser un certain nombre de réacteurs extrinsèques (TabRefRExtrinseques).

```
class RIntrinseques
{
        int             CodeReacteur;
        char*           Nom;
        RExtrinseques*  TabRefRExtrinseques [NbMaxRExtrinseques];
        void            (*FonctionParametres) (ObjetsApplications*, Proprietes*);
public:
        méthodes ...
};
```

2.4. Les transmuteurs.

Comme les réacteurs, les transmuteurs sont définis par un identifiant unique (Code). Par contre, il est nécessaire de préciser deux fonctions. La première représente le domaine de transmutation (FonctionTesteTransmutation), elle teste si l'objet doit transmuter. La seconde effectue effectivement la transmutation (FonctionCreeObjTransmute).

```
class Transmuteurs
{
      int                   Code;
      char*                 Nom;
      char*                 NomTypeTransmute;
      Booleen               (*FonctionTesteTransmutation) (ObjetsApplications*,
                                                           ModeleObjetsApplication*);
      ObjetsApplications* (*FonctionCreeObjTransmute)  (ObjetsApplications*,
                             ModeleObjetsApplication*, ModeleObjetsApplication*);
public:
      méthodes ...
};
```

Dans notre implémentation, nous avons développé uniquement les transmuteurs extrinsèques.

2.5. Les producteurs.

Tout producteur est construit par rapport aux comportements qu'il utilise. On indique en particulier, les réacteurs intrinsèques (TabRefRIntrinseques), les transmuteurs (TabRefTransmuteurs) et les réacteurs extrinsèques (TabRefRExtrinseques). Il faut également indiquer la fonction de création de la représentation canonique (CodeInitObj) et la fonction de mise à jour (CodeMAJ).

```
class Producteurs
{
          char*           Nom;
          TabAttributs    *TabAttribut;
          RIntrinseques*  TabRefRIntrinseques [NbMaxRIntrinseques];
          RExtrinseques*  TabRefRExtrinseques [NbMaxRExtrinseques];
          Transmuteurs*   TabRefTransmuteurs [NbMaxTransmuteurs];
          int             CodeMAJ, CodeInitObjet;
public:
          méthodes ...
};
```

Les attributs et leurs contraintes respectives sont précisés dans un tableau d'attributs (TabAttribut).

3. LA MAQUETTE.

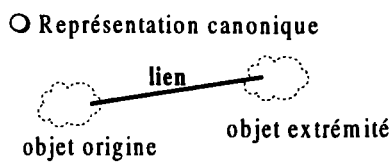
L'objectif est de décrire une application permettant de visualiser le formalisme graphique de description du dialogue, en proposant des créations graphiques de haut niveau (détections, aides ...). La description de l'application passe par la spécification du modèle des objets. Chaque objet est analysé du point de vue comportements : cette vision est celle de l'opérateur non informaticien. Il s'agit de décrire comment l'on souhaite voir fonctionner l'application. Les comportements facilitent cette description car ils imposent une certaine discipline. Les types de comportements répondent à des questions sur l'organisation de l'application :

- les producteurs : quand et comment créer les objets ?
- les transmuteurs extrinsèques : quand et comment transformer les objets ?
- les réacteurs intrinsèques : quand et comment détecter des propriétés ?
- les réacteurs extrinsèques : quand et comment montrer les objets ?

En plus des entités du formalisme, nous avons choisi d'intégrer un objet lien dont le but est de décrire à une transmutation près toutes les relations entre les entités (séquence, contrainte, ...). C'est cet objet que nous analysons tout au long de cette annexe.

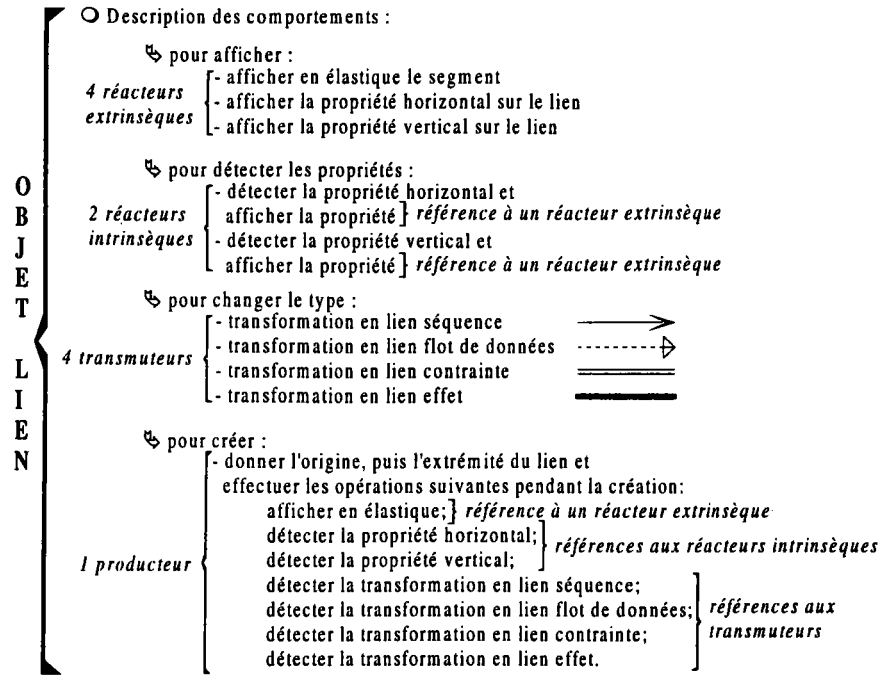
Les objets sont tout d'abord décrits comme des objets d'application. Nous créons donc une classe dérivée de la classe *ObjetsApplications* et qui contient la représentation canonique du lien.

```
class ObjetsLiens:
 public ObjetsApplication
{
        int         ObjetOrigine, ObjetExtremite;
        Points*     PointOrigine, PointExtremite;
public:
        méthodes ...
};
```



En plus des coordonnées, le lien est représenté par les objets origine et extrémité.

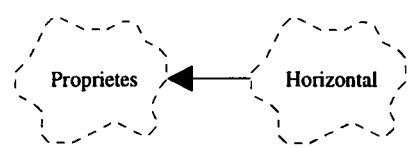
Les objets sont ensuite spécifiés à travers leurs différents comportements. Pour le lien, on recherche les comportements significatifs afin de construire une instance de la classe *ObjetsSpecifies* et contenant la spécification.



Le seul producteur indique les attributs permettant de définir le lien i.e. le type des objets lui servant d'origine et celui des objets lui servant d'extrémité, ce dernier devant satisfaire une contrainte de validité avec l'objet origine (voir chapitre 3 pour la validité des relations du formalisme). Pendant la création avec ce producteur, des aides vont intervenir (affichages, détection de propriétés et transformation en liens typés). On remarque les quatre relations du formalisme apparaissent après transmutation.

Dans cette spécification, on peut remarquer la propriété "horizontal". Comme toute propriété, elle est créée à partir de la classe *Proprietes*. Nous créons une classe dérivée de la classe *Proprietes* qui contient les données nécessaires à sa détection. La seule donnée est le vecteur directeur (V). Le réacteur intrinsèque associé à un objet désirant supporter cette propriété n'aura qu'à indiquer cette donnée pour en permettre la détection.

```
class Horizontal:
  public Proprietes
{
            Vecteurs*  V;
public:
            méthodes ...
};
```



Tous les objets du formalisme ont été décrits de cette façon ainsi que les propriétés "horizontal", "vertical" et "coller contre". Cette dernière propriété est illustrée dans le paragraphe 4.2.

4. LA GÉNÉRATION.

Après la spécification du modèle, la génération du dialogue est déclenchée et le modèle est analysé .

4.1. La bibliothèque de présentation.

C'est tout d'abord une véritable arborescence de menus de création qui est déduite (figure D.1) : chaque objet possède un menu général composé de sous menus précisant les différents modes de création possibles. Ces modes proviennent des producteurs de l'objet.

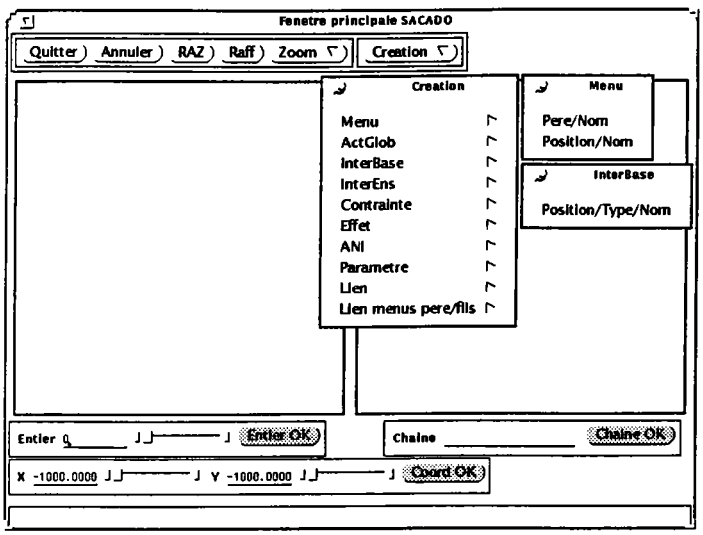
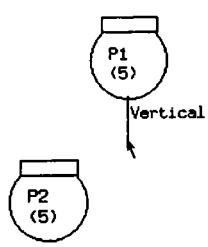


Figure D.1. *La bibliothèque de présentation déduite du modèle.*

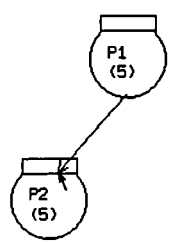
Par exemple, un objet *Menu* peut être créé de deux façons : *Pere/Nom* ou *Position/Nom*. Au contraire, un objet *InterBase* ne possède qu'un seul producteur, *Position/Type/Nom*. À chacun de ces menus, la génération associe l'action globale générée à partir du producteur correspondant.

4.2. La bibliothèque des actions globales.

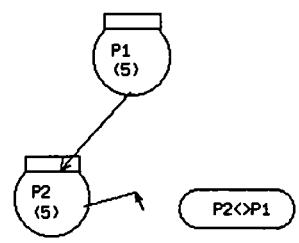
Une analyse des comportements spécifiés pour chaque objet permet la génération de l'action globale qui correspond au dialogue de création. Pour mettre en valeur le type de dialogue généré, on se place désormais du côté manipulation de l'application. On choisit l'exemple de la définition d'une action globale de création d'un segment par deux points différents. Chaque image correspond à une capture d'une portion de la fenêtre graphique de définition des actions globales.



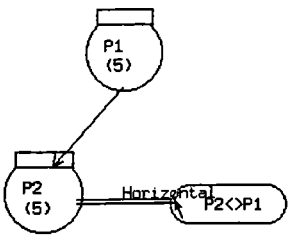
1. Création d'un objet lien vertical.



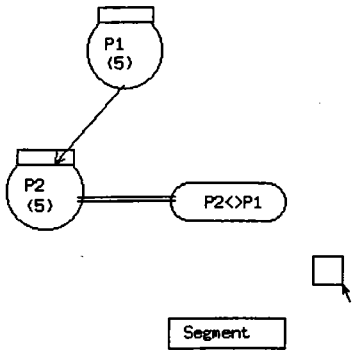
2. Transmutation en un objet lien séquence.



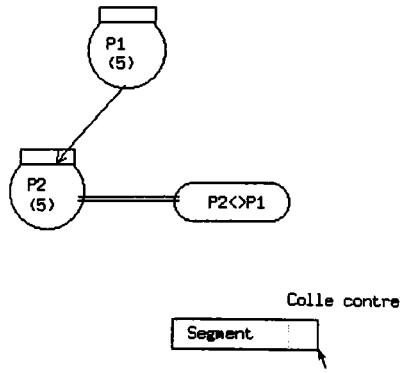
3. Création d'un objet lien.



4. Transmutation en un objet lien contrainte horizontal.



5. Création d'un objet paramètre.



6. Création d'un objet paramètre collé contre un objet action non interactive.

On suppose que l'opérateur a déjà créé deux objets interaction de base et se trouve dans une action de création d'un objet lien. Les deux premières étapes montrent comment l'application gère cette création grâce au producteur défini pour le lien. À partir des informations incluses dans la définition de l'objet lien et décrites précédemment, le dialogue a été généré pour permettre la saisie des deux attributs. Dès que l'opérateur fournit un attribut demandé, une mise à jour de la représentation canonique du lien est automatiquement déclenchée.

L'étape n°1 montre que le dialogue déduit est plus riche qu'une simple saisie des attributs : l'opérateur peut voir le lien avant sa création définitive (affichage en élastique) et le système l'aide à créer un lien vertical. Ces deux aides résultent de l'analyse de deux autres comportements qui ont été associés au producteur pour intervenir pendant la création du lien. Le premier ne décrit qu'un affichage particulier, il ne concerne que l'apparence du lien : c'est un réacteur extrinsèque (dans l'implémentation, il est entièrement défini par une fonction définissant l'affichage à réaliser). Le second comportement permet de détecter une propriété sur le lien : c'est un réacteur intrinsèque. Référencer un tel réacteur intrinsèque pour le producteur indique que le dialogue va devoir faire un traitement supplémentaire durant la création (calcul de la donnée nécessaire à la propriété). L'affichage de la propriété est l'œuvre d'un réacteur extrinsèque associé au réacteur intrinsèque.

L'étape n°2 concerne toujours la création du lien, mais elle met en évidence un troisième type de comportement : le transmuteur extrinsèque. Lorsque l'extrémité du lien est une interaction, le lien devient un lien séquence. Ce dialogue est déduit par l'analyse du transmuteur qui contient tous les tests nécessaires à la détection ainsi que l'opération permettant de réaliser le changement de type.

Les étapes n^{os} 3 et 4 montrent encore la création d'un lien, mais elles proposent d'autres combinaisons de comportements décrits également dans le producteur : la transmutation en lien contrainte et la détection de la propriété horizontal. À tout moment les comportements associés au producteur sont déclenchés. Ils ne se réalisent que si la représentation du lien est satisfaisante.

À l'étape n°5, l'opérateur ajoute un objet paramètre à l'action non interactive "Segment". Nous avons choisi de décrire cette création car elle montre la détection d'une propriété binaire ("Coller contre"). Tout comme le lien, le paramètre est affiché en élastique pendant la création grâce à un réacteur

extrinsèque associé au producteur. L'étape n°6 permet de constater que le dialogue de création intègre la détection de la propriété. Pour se faire, les deux objets mis en jeu dans la propriété (paramètre et action non interactive) intègrent un réacteur intrinsèque. Le programmeur d'interfaces, pour indiquer qu'un objet peut recevoir une propriété, n'a eu qu'à décrire les réacteurs intrinsèques i.e. l'évaluation des données et les éventuels affichages (réacteurs extrinsèques) associés à la détection. Toute la gestion du dialogue qui accompagne la détection de la propriété (savoir comment et à quel moment faire les différents traitements) est générée.

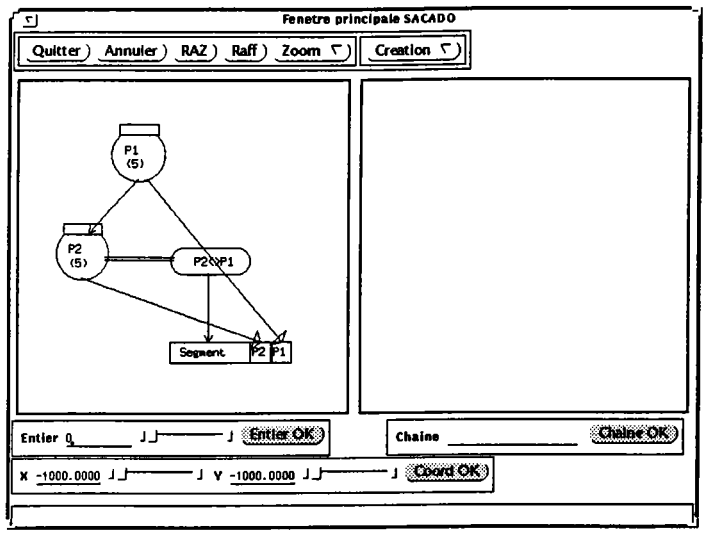


Figure D.2. *L'action globale de création d'un segment terminée.*

La figure précédente visualise le type d'action globale que l'on peut spécifier avec l'application. On peut décrire graphiquement un dialogue à l'aide du formalisme en utilisant l'application.

4.3. La bibliothèque des interactions.

Au cours de la génération des menus et des dialogues associés aux producteurs, il est possible de compléter la bibliothèque des interactions. Pour chaque objet analysé, il suffit d'ajouter la ligne suivante dans la bibliothèque :

```
COMPATIBILITE : INTERFACE Nom_Menu EST Local DE INTERACTION Type_Objet;
```

Dans ce cas, *Nom_Menu* représente le nom du menu de création et *Type_Objet* est le type de l'objet (code associé à l'objet spécifié). On obtient par exemple pour notre implémentation :

```
COMPATIBILITE INTERACTION :

COMPATIBILITE : INTERFACE "Menu" EST Local DE INTERACTION 10;
COMPATIBILITE : INTERFACE "ActGlob" EST Local DE INTERACTION 11;
...

FIN
```

5. CONCLUSION.

La maquette est opérationnelle. Si l'on observe l'implémentation d'un point de vue conceptuel, la description des comportements ne semble pas complexe. Elle correspond à la vision que l'on a du dialogue. Pour détecter une propriété sur un objet, on crée un réacteur intrinsèque lié à la propriété et qui contient la description de l'évaluation des données utilisées pour vérifier la propriété et dépendant de l'objet. On reste donc suffisamment éloigné des concepts du dialogue à utiliser tout en restant proche de la représentation que l'opérateur se fait du dialogue.

On peut considérer que le dialogue obtenu est satisfaisant. On a pu constater à l'aide de l'exemple que le dialogue ne correspond pas à une simple saisie des attributs comme on pourrait le penser. On aboutit à des dialogues de plus haut niveau avec l'intégration d'aides tels que des affichages particuliers (grâce aux réacteurs extrinsèques), des détections de propriétés (grâce aux réacteurs intrinsèques) et des changements de type (grâce aux transmuteurs). Le dialogue obtenu est donc plus que satisfaisant.

Un intérêt supplémentaire apparaît sur le plan de l'extension. Il est tout à fait possible de modifier l'application en spécifiant de nouveaux objets. Le dialogue pourra encore être généré. En faisant abstraction de l'absence d'outils de description des comportements, l'implémentation a prouvé que générer du dialogue n'est pas une utopie. Il reste encore des zones d'ombres à éclaircir, mais les résultats sont prometteurs car avec peu de moyens on obtient l'application graphique interactive que l'on désirait initialement.

CONTRIBUTION POUR UNE NOUVELLE APPROCHE DU DIALOGUE HOMME-MACHINE EN C.F.A.O.

Ces dernières années, on est passé d'un dialogue rudimentaire à une forme de dialogue plus évoluée où l'utilisateur interagit directement dans un espace de travail reconstitué graphiquement. La conception de telles interfaces est devenue un des problèmes clés pour le développement des Applications Graphiques Interactives (AGI).

Nous montrons que de nombreux modèles ont été introduits pour aider à la prise en charge de l'utilisateur et à la conception des interfaces utilisateur. Après une étude de ces différents modèles et des outils de développement couramment rencontrés, nous abordons notre domaine d'application, la C.F.A.O. (Conception et Fabrication Assistées par Ordinateur). Pour répondre aux difficultés inhérentes à ce domaine, un nouveau modèle d'interaction est introduit. Il prend en compte les intentions de l'utilisateur à travers une primitive unique de dialogue appelée INTERACTION qui assure une grande liberté à l'utilisateur tout en garantissant la cohérence des dialogues engagés. Autour de ce modèle d'interaction, nous introduisons deux modèles de dialogue. L'un est textuel et est destiné à des utilisateurs spécialisés alors que l'autre est basé sur un formalisme graphique pour des utilisateurs moins expérimentés.

Pour aller plus loin dans l'aide apportée au développement, nous proposons d'utiliser les connaissances incluses dans le modèle manipulé par l'utilisateur. Par une transformation adéquate de la connaissance liée aux objets de ce modèle, il est possible de déduire en grande partie l'interface utilisateur. Dans la mesure où cette interface déduite se présente sous le même formalisme graphique, il sera toujours possible de l'adapter ou de l'enrichir. Cette dualité dans le développement de l'interface nous permet d'envisager qu'un utilisateur non informaticien puisse concevoir sa propre interface voire même l'architecture complète de son logiciel.

Mots-clés : C.F.A.O., A.G.I., approche modèles, P.O.O., interfaces homme-machine, interaction, dialogue.