Laboratoire d'Informatique Scientifique et Industrielle

École Nationale Supérieure
de Mécanique et d'Aérotechnique

Université de Poitiers
Faculté des Sciences Fondamentales et Appliquées

Spécialité : Informatique

Ingénierie des systèmes interactifs : vers des méthodes formelles intégrant l'utilisateur

Patrick Girard
Maître de Conférences

Présentation des travaux en vue de l'obtention de
l'Habilitation à diriger les recherches

soutenue le 27 Janvier 2000, ENSMA, FRANCE

Jury :
M. Beaudouin-Lafon (rapporteur), Professeur à l'Université de Paris-Sud, Orsay,
J. Coutaz (rapporteur), Professeur à l'Université de Grenoble,
B. David, Professeur à l'École Centrale de Lyon,
C. Kolski, Professeur à l'Université de Valenciennes,
H. Lieberman (rapporteur), Research Scientist au Massachusetts Institute of Technology, Cambridge, USA,
G. Pierra, Professeur à l'ENSMA.

Sommaire

Chapitre 1
Introduction

L'Interaction Homme-Machine (IHM) est un domaine en forte expansion. De nombreuses associations se sont constituées dans l'objectif de promouvoir la recherche en IHM, comme SIGCHI (Special Interest Group on Computer Human Interaction) dans le cadre d'ACM (Association for Computing Machinery), le groupe TC 2.7 (User Interface Engineering) de l'IFIP (the International Federation for Information Processing), le HCI Group du BCS (British Computer Society) ou encore l'AFIHM (Association Francophone pour l'Interaction Homme-Machine). Depuis la fin des années 1980, le nombre de conférences qui lui sont totalement consacrées a considérablement augmenté. À côté des conférences généralistes que sont ACM CHI (Conference on Human Factors in Computing Systems), IFIP Interact et EHCI (Engineering of Human Computer Interaction) au plan international, BCS HCI en Angleterre (Human Computer Interaction), AFIHM IHM (Ingénierie de l'Interaction Homme-Machine) et Ergo-IA (Ergonomie et Informatique Avancée) dans l'espace francophone, des « workshops » et autres « symposiums » plus ciblés ont vu le jour, comme ACM UIST (Symposium on User Interface and Software Technology) ACM DIS (Design of Interactive Systems), et CADUI (Computer-Aided Design of User Interfaces) en ce qui concerne un aspect plutôt orienté outils, ou encore DSV-IS (Eurographics Workshop on Design, Specification and Validation of Interactive Systems) et BCS FAHCI (Formal Aspects of Human Computer Interaction) sur les aspects plus formels. Cette liste n'est pas exhaustive…

Comme certains intitulés le laissent clairement entendre, le domaine de l'Interaction Homme-Machine est fortement pluridisciplinaire, puisque, en plus de l'informatique et de la technologie principalement électronique, il fait intervenir la psychologie, l'ergonomie, les sciences cognitives, la linguistique ou encore les arts graphiques. Située par essence à l'interface entre les mondes informel de l'utilisateur et formel de l'informatique, l'IHM constitue une part de plus en plus importante du coût de développement des applications [Myers, Hollan, & Cruz 1996]. Cependant, malgré l'effort de recherche qui lui est consacré, on est encore loin de disposer de méthodes et d'outils adaptés à toutes les formes d'applications interactives. La nature pluridisciplinaire du domaine, mais aussi la très forte évolutivité des dispositifs et techniques d'interaction contribuent à remettre très régulièrement en question les avancées pourtant nombreuses.

Nous nous situons clairement dans la partie informatique du domaine, et plus particulièrement dans l'aspect logiciel. Comme nous le verrons tout au long de ce mémoire, nos travaux ont donné lieu à de nombreux développements pour valider les idées émises. Quelques-uns ont même conduit à des outils opérationnels, pour une utilisation soit académique, soit industrielle. Cela ne nous a pas empêché d'étudier les principes généraux des Sciences Cognitives qui trouvent leur application dans les IHM, et la suite de ce mémoire fera abondamment référence au modèle du processeur humain [Card, Moran, & Newell 1983] et à la théorie de l'action de Norman [Norman 1986]. De la même façon, nous nous appuierons sur le corps de discipline bien établi par les livres références du domaine, qu'ils soient en français [Barthet 1988 ; Coutaz 1990 ; Kolski 1997] ou en anglais [Bass & Coutaz 1991 ; Dix, et al. 1998 ; Olsen 1998 ; Shneiderman 1998]. Un autre très fort point d'ancrage de nos travaux réside dans les grands principes du génie logiciel [Gaudel, et al. 1996 ; Pierra 1991], qui ont toujours conduit notre démarche de développement, et auxquels nous ferons souvent implicitement référence.

Pour ce mémoire, nous avons choisi de réaliser une synthèse des travaux que nous avons menés. Malgré la diversité des publications auxquelles ils ont donné lieu depuis dix ans, il est aisé de mettre en avant le fil conducteur qui les a dirigés et qui se résume à la première partie du titre de ce mémoire : l'ingénierie des systèmes interactifs. Nous décrivons l'essentiel de nos travaux en trois chapitres principaux, et exposons ensuite notre programme de recherche.

Nous avons commencé nos travaux de recherche dans le domaine de la conception technique, avec l'objectif de permettre à un utilisateur d'application de Conception Assistée par Ordinateur (CAO) de « programmer », plus ou moins implicitement, dans l'interface. Ceci nous a conduit à établir de nombreux parallèles avec un thème émergent du domaine des IHM, appelé globalement « Example-based programming » (programmation sur exemple), puis « Programming by Demonstration ». Nous décrivons les différents travaux que nous avons menés dans cette direction tout au long des dix dernières années dans le Chapitre 2. Cet axe est demeuré central par rapport à nos préoccupations, d'où l'importance du développement qui lui est consacré.

Parce que les travaux menés ont nécessité beaucoup de développements, nous nous sommes rapidement intéressé aux outils. À partir de notre domaine initial, la CAO, nous avons pu isoler des besoins spécifiques et développer un modèle de dialogue particulier pour lequel un modèle d'architecture a été conçu. Nous avons ainsi exploré l'espace des outils de développement des interfaces, domaine auquel nous avons contribué. L'ensemble de ces travaux est décrit dans le Chapitre 3.

Comme cela est fréquent en informatique, nous avons souvent défini et utilisé des langages spécifiques : langages exécutables par le biais de la programmation sur exemple, ou langages de spécification de dialogues, nous avons défini la syntaxe de langages, et les avons utilisés pour générer tout ou partie d'applications. Il était donc naturel de se pencher un peu plus sur la sémantique de ces langages, et sur les raisonnements que l'on peut leur associer. C'est ainsi que nous avons abordé plus récemment les méthodes formelles dans le cadre des IHM. Le Chapitre 4 décrit des travaux que l'on peut qualifier de préliminaires dans ce domaine, même s'ils ouvrent de nombreuses perspectives que nous entendons creuser à l'avenir.

L'exploration de trois sous-domaines connexes, quoique largement distincts, de l'ingénierie de l'IHM nous a permis d'acquérir la certitude que les travaux actuels ne sont pas encore allés assez loin sur le chemin de l'intégration de l'utilisateur dans le processus de réalisation des applications interactives. C'est la thèse que nous défendrons dans le Chapitre 5 qui développe notre programme de recherches.

Dans un domaine en pleine évolution qui n'a vraisemblablement pas encore atteint sa pleine maturité, il est naturel de se trouver confronté à des problèmes de vocabulaire, phénomène d'autant plus important que le poids de la langue anglaise est considérable en informatique. Nous avons essayé, tout au long de ce mémoire, d'apporter des définitions claires aux termes et expressions que nous employons. Lorsque cela nous semblait nécessaire, nous avons recherché le sens primitif des expressions en langue anglaise pour proposer une traduction idoine en français. Par exemple, nous refusons d'utiliser la traduction mot à mot de l'expression « Programming by Demonstration » en « Programmation par Démonstration », l'acception française du mot « démonstration » ne le permettant pas.

Dans la suite de ce mémoire, nous emploierons le sigle IHM pour désigner le domaine général qui nous intéresse. Le plus souvent, pourtant, ce n'est pas d'Interaction Homme-Machine que nous parlerons, et encore moins d'Interfaces Homme-Machine comme l'on pouvait dire il y a quelques années. Dans la majorité des cas, nous parlerons d'Ingénierie de l'Interaction Homme-Machine, thème central de nos travaux. De la même façon, nous parlerons peu d'interfaces graphiques, mais plus généralement d'Applications Graphiques Interactives (AGI), concept qui nous paraît regrouper à la fois les aspects communs aux interfaces graphiques (le standard aujourd'hui), et le domaine des applications utilisant intensivement la représentation graphique des objets composant leur noyau fonctionnel.

L'ordre des chapitres reflète la chronologie de nos travaux, mais chacun d'eux suit une présentation standard. Après une courte synthèse des travaux concernant chaque sous-domaine, nous détaillons les études que nous avons réalisées. Les publications que nous avons faites ainsi que celles qui relèvent de notre encadrement sont mises en relation avec les résultats obtenus. Puis, une troisième partie expose quelques-unes des perspectives ouvertes.

Chapitre 2
Programmation sur exemple / « Programming by demonstration »

Nous avons débuté nos travaux par la résolution d'un problème concret : le paramétrage en Conception Assistée par Ordinateur. Ce faisant, nous avons abordé un domaine en émergence au début des années 1990, que les anglophones appellent « end-user programming ». Au fil des projets, nous avons ainsi évolué vers des thèmes plus proches de l'interaction homme-machine. Dans la section suivante, nous utilisons principalement les expressions anglaises sans les traduire. Nous exposons dans la section 2.2.1 notre proposition de traduction de ces termes, basée sur une classification du domaine qui nous intéresse le plus, celui des environnements de programmation.

2.1 Le domaine de recherche

En mars 1990, sortait le premier numéro d'une revue publiée par les « Academic Press », et intitulée « Journal of Visual Languages and Computing ». Cet événement marquait l'émergence de nouvelles voies en vue de simplifier la tâche complexe que constitue la programmation par l'usage de la visualisation. L'ensemble de ces travaux, que l'on peut regrouper sous le vocable de « end-user computing », visait globalement, au prix d'un usage intensif du graphique, à permettre à un utilisateur de système interactif de réaliser des « programmes ».

Le précurseur en la matière semble être Shneiderman qui, en 1983, après de nombreux travaux consacrés à l'usage du graphique et de l'interaction graphique dans diverses tâches (programmation, accès aux bases de données…), effectue une première étude de la « manipulation directe » : « Direct manipulation : a step beyond programming languages » [Shneiderman 1983]. Force exemples à l'appui, il pose les bases de ce que certains appellent déjà la programmation visuelle (« Visual programming » [Macdonald 1982]). Si Raeder [Raeder 1985] propose un survol des techniques de programmation graphique utilisées dans cet axe émergent, la codification précise est à mettre au bénéfice des travaux de Chang [Chang 1986] et Shu [Shu 1986] pour ce qui concerne le seul aspect graphique, et de ceux de Halbert [Halbert 1984] et Myers [Myers 1986 ; Myers 1988 ; Myers 1990b ; Myers 1990c] pour ce qui concerne la notion d'exemple d'exécution en tant qu'outil d'aide à la définition d'un programme.

Nous rappelons tout d'abord les points essentiels de ces classifications, respectivement dans le domaine de la « Visual Programming » et de l' « Example-Based Programming ». Dans un deuxième temps, nous développons les définitions de la « Programming by Demonstration ».

2.1.1 Les fondements : « Visual Programming »

La « Visual Programming » a littéralement explosé au milieu des années 1980. L'augmentation de puissance des machines a rendu possibles nombre de travaux demeurés jusqu'alors très théoriques. Les deux recueils de Glinert [Glinert 1990] constituent ainsi un formidable éventail de systèmes très divers. Mais c'est en 1986 que les définitions et classifications ont été proposées. Nous survolerons ici celles de Shu [Shu 1986] et Chang [Chang 1986] avant de glisser vers des notions plus ciblées.

Shu [Shu 1986] décrit comme suit les avantages que présente l'utilisation de l'image dans l'activité de programmation :

1. Les gens préfèrent en général les images aux mots.
2. Les images constituent un moyen de communication plus efficace que les mots. Elles peuvent contenir plus de signification dans une unité d'expression plus concise.

3. Les images ne sont pas sujettes aux barrières de la langue. Elles sont comprises par les peuples, quelle que soit cette dernière.

Quoique largement discutables, ces affirmations reflètent la motivation principale de la programmation visuelle. Il s'agit de remplacer la nature textuelle des programmes par l'utilisation de l'image. Cette utilisation de l'image ne se concrétise cependant pas de façon identique dans les différents systèmes proposés, ce qui justifie les classifications précédemment évoquées.

Chang [Chang 1986] classe ainsi les « Visual Languages » selon deux critères orthogonaux :

- Les objets manipulés sont naturellement visualisables (les différents éléments d'une image provenant d'une caméra, par exemple) ou ils ne le sont pas (leur visualisation passe alors par des conventions plus ou moins naturelles pour l'utilisateur : c'est la notion « d'object icons »), et ils possèdent alors une représentation visuelle schématique.
- Les structures de programmes sont représentées de façon textuelle (représentation linéaire classique) ou elles sont associées à une représentation graphique (« process icon »).

Cette démarche permet de définir quatre classes de « Visual Languages » en croisant ces deux critères. Cette classification apparaît cependant insuffisante dans la mesure où elle ne porte que sur la phase de construction statique du programme sans s'intéresser aux représentations visuelles susceptibles d'être créées par le système à l'issue de chaque phase constructive de façon à visualiser le programme et/ou son exécution.

Shu [Shu 1986] distingue, dans ce qu'il nomme « Visual Programming », les environnements visuels (« Visual Environments ») ou à visualisation de programmes (« Program visualization ») qui permettent de *représenter, a posteriori*, tout ou partie du programme de façon graphique, et les « Visual Languages » qui permettent de *construire* le programme, par interaction graphique.

Si ces études sont intéressantes pour tenter de classifier par rapport à l'aspect visuel la multiplicité des systèmes graphiques qui apparaissent chaque jour, elles ne peuvent guère être utilisées pour évaluer l'aide réelle qu'un tel système peut apporter dans l'activité de programmation. Le fait, en particulier, que l'utilisateur doive, graphiquement ou non, manipuler des variables, ce qui constitue la première difficulté de la programmation, ou qu'il puisse se contenter de manipuler des valeurs, ce qui lui est beaucoup plus naturel, n'apparaît pas dans ces classifications. Ce distinguo important sera analysé par Halbert [Halbert 1984] et Myers [Myers 1990c] à travers la notion d'*exemple*.

2.1.2 L'émergence : « Example-Based Programming »

Utiliser des exemples d'exécution pour concevoir un programme est l'étape suivante permettant d'aller vers un environnement de développement utilisable par un utilisateur final non-informaticien. Au lieu de sélectionner des fonctions et de définir des variables sur lesquelles les fonctions s'appliqueront, l'utilisateur utilise des fonctions sur des valeurs, qui se comportent comme les variables d'un programme. L'idée est ainsi d'éviter le niveau d'abstraction des variables en autorisant l'utilisateur à manipuler des exemples, correspondant à des valeurs spécifiques de ces variables.

Halbert [Halbert 1984] est le premier auteur à avoir recensé et comparé les diverses approches concourant à permettre à un utilisateur final (« end-user »), non-programmeur, de créer des programmes. L'idée générale de ces travaux est qu'il est plus facile de comprendre le programme que l'on cherche à écrire si celui-ci s'exécute parallèlement à sa construction.

À partir de ses travaux du début des années 1980, Lieberman [Lieberman 1990] définit le premier en français l'expression « programmation par l'exemple ». Se basant sur l'idée qu' « une interface graphique est un langage qui permet à l'utilisateur de demander à l'ordinateur de réaliser ses intentions » et prenant comme point de départ l'idée d'utiliser des exemples pour enseigner un processus à l'ordinateur dans un environnement de programmation, Lieberman définit ainsi le

système Tinker [Lieberman 1982] qui possède déjà quelques-unes des caractéristiques essentielles des systèmes de programmation par l'exemple.

À la suite de Halbert, et en marge de ses travaux sur le système Peridot sur la programmation d'interface utilisateur, Myers analyse dès 1986 le domaine récent de la « Visual Programming ». S'appuyant sur les définitions de Halbert pour la partie « exemple », il propose une taxinomie qu'il fait évoluer au fil des publications [Myers 1986 ; Myers 1988 ; Myers 1990b ; Myers 1990c]. Son idée majeure est de classer toutes les approches de la programmation selon trois critères principaux, orthogonaux, et selon un critère secondaire.

Myers part de la définition de la programmation, tirée du « Dictionary of computing » [Oxford 1983] : « un programme est un ensemble d'instructions qui peuvent être transmises à un ordinateur et utilisées pour commander le comportement de ce système ». Il adjoint néanmoins à cette définition la nécessité de pouvoir comporter des *variables*, des *conditions* et des *itérations*, même de façon implicite. Ses trois critères principaux sont basés sur les définitions suivantes :

- **Compilé / Interprété :** Un système compilé se caractérise par un temps d'attente important avant que les instructions puissent s'exécuter, en raison de leur traduction en représentation de bas-niveau, en mode différé (« batch »). Un système interprété permet une exécution des instructions au fur et à mesure de leur création (Myers reconnait néanmoins l'existence d'un *continuum* plus qu'une dichotomie entre ces deux modes).
- **« Visual Programming » :** La définition utilisée par Myers pour caractériser la « Visual Programming » est plus restrictive que celles évoquées plus haut. En effet, elle exclut non seulement les systèmes ne présentant qu'une visualisation graphique des programmes ou des données, mais également les langages textuels manipulant des données graphiques. Myers n'admet le qualificatif de « Visual Programming » que dans le cas où les systèmes permettent à l'utilisateur de définir des programmes « in a two - or more - dimensional fashion ».
- **« Example-Based Programming » :** Il s'agit ici de regrouper tous les systèmes qui permettent au programmeur d'utiliser un exemple de données d'exécution pendant la phase de programmation. Un système est « Example-Based », si un exemple d'exécution du programme se déroule parallèlement à sa construction, il n'est pas « Example-Based » dans le cas inverse.

Dans le cadre de l' « Example-Based Programming », un critère secondaire doit être pris en compte, défini par Halbert [Halbert 1984] sous le terme d'*inférence* (« Inferencing »). Pour cet auteur, un système est dit « avec inférence » s'il est capable de « déduire, des actions de l'utilisateur, un programme », sans que ce dernier ait effectivement dit comment le faire. En 1988, Myers cherche à préciser ce critère. Il parle alors d'inférence plausible (« Plausible Inferencing » ou « abduction ») pour caractériser les systèmes qui génèrent des explications ou des généralisations basées sur des informations limitées. Ces explications peuvent se révéler incorrectes, et ces systèmes prennent souvent l'initiative d'un dialogue avec l'utilisateur pour lever les ambiguïtés. Ceci l'amène à distinguer ce qu'il appelle « Programming-With-Examples », *sans inférence plausible*, encore caractérisé par Halbert par la formule « Do What I Did » (« Fais ce que j'ai fait »), et « Programming-By-Example », *avec inférence plausible*, caractérisée toujours par Halbert par la formule « Do What I Mean » (« Fais ce que je pense ») [Halbert 1984].

On peut illustrer cette classification avec quelques systèmes créés antérieurement :

- *Compilé, VP, NOT EBP :* Pro-Blox [Glinert 1987] est un environnement de programmation qui permet, à l'aide de « pièces de puzzle » représentant les structures de contrôle du langage PASCAL, de créer graphiquement des programmes qui peuvent être ensuite compilés, selon les règles de ce langage. Outre la représentation graphique, l'emboîtement des pièces a le mérite d'interdire les fautes de syntaxe et de permettre une meilleure visualisation de la structure du programme.
- *Interprété, VP, NOT EBP :* Labview® (National-Instruments) est un environnement de programmation graphique qui permet surtout de simuler ou de contrôler des appareils de mesure et d'effectuer du traitement du signal. Les programmes sont représentés grâce à

un formalisme de flots de données. Si Labview® allie la représentation purement graphique (les icones connectées les unes aux autres) et une forme textuelle, cela n'a rien d'obligatoire, et l'on peut programmer totalement graphiquement, en incluant toutes les structures de contrôle. Une particularité intéressante de Labview® réside dans la définition, indépendamment de celle du programme, de l'interface utilisateur de ce dernier, sous une forme identique à celle qu'elle aura en exécution. Comparé au système précédent, Labview® possède l'avantage d'être interprété au fur et à mesure de la connexion des icones, une icone de la barre d'état visualisant à tout moment la correction syntaxique (ou non) du diagramme (c'est-à-dire le programme).

- *Interprété, VP, EBP, sans inférence :* SmallStar [Halbert 1984] permet de construire des programmes avec exemple dans l'environnement Star® (Xerox). Semblable à première vue aux utilitaires de « Macros » du type de AppleScript® sur MacIntosh® (Apple), il diffère de ces derniers par la possibilité de définir des variables, et donc de construire de véritables programmes. Pour ce faire, l'utilisateur doit explicitement exprimer la nature des objets qu'il manipule (constante, variable, paramètre). Une représentation semi-textuelle, semi-iconique du programme généré permet à l'utilisateur de rajouter des structures de contrôle.
- *Interprété, VP, EBP, avec inférence :* Peridot [Myers 1988 ; Myers 1990a ; Myers & Buxton 1986] est un système de création d'interface « by example ». À partir de valeurs données pour l'exemple, l'utilisateur crée l'interface utilisateur telle qu'elle apparaîtra lors de l'utilisation ; Peridot génère un code InterLisp qui peut ensuite être modifié pour y ajouter des structures de contrôle. Le fonctionnement même de l'interface utilisateur n'est pas décrit explicitement par le programmeur, et l'inférence est utilisée abondamment pour fournir les contraintes géométriques et détecter les actions sur des listes d'entités.

2.1.3 La généralisation : « Watch What I Do: Programming by demonstration »

En mai 1991, au cours de la conférence SIGCHI'91, à La Nouvelle-Orléans, s'est tenu un premier « Workshop » sur les langages pour développer les interfaces utilisateurs [Myers 1992a] au cours duquel un groupe de travail a été constitué sur le thème « End-User Computing ». Dans le rapport du groupe [Myers 1992b], les participants relèvent le fait que les utilisateurs de systèmes informatiques sont de plus en plus demandeurs de possibilités de programmation. Ils concluent cependant en remarquant que l'objectif principal de la programmation pour les utilisateurs est d'accomplir leur tâche de façon plus efficace, et surtout pas de créer de la complexité…

L'année suivante, s'est tenu chez Apple Computer un nouveau « Workshop » où les pionniers du domaine de l' « Example-Based Programming » ont effectué nombre de démonstrations de leurs systèmes. En dehors du recueil, et souvent de la réécriture, de leurs travaux dans un ouvrage qui fait aujourd'hui référence [Cypher 1993c], les participants ont affiné les définitions et proposé une nouvelle expression, « Programming by Demonstration » (PbD), qui est aujourd'hui largement acceptée. Dans un deuxième temps, ils ont recensé les différents éléments caractérisant les systèmes de PbD, et proposé une grille d'évaluation de ces systèmes.

2.1.3.1 Définition

Pour Cypher [Cypher 1993b], la PbD est une extension du concept des enregistreurs de macro. Ces derniers sont à même d'enregistrer les actions de l'utilisateur puis de les rejouer à la demande. Cependant l'exécution des macros ainsi réalisées ne prend pas en compte le contexte de création ou d'exécution. L'enregistrement est au niveau des commandes et des clics souris. Dans le cas de la PbD, le système produit une généralisation des actions enregistrées. Durant la conception d'un programme, le système analyse les entrées de l'utilisateur, et construit le programme à même de générer l'exemple, ainsi que les variantes de l'exemple. Cette analyse peut se faire sous la forme d'un

enregistrement immédiat de la séquence des actions ou par un mécanisme d'abstraction sur l'ensemble de l'exemple.

L'un des grands avantages de la PbD sur la programmation textuelle conventionnelle est qu'il s'agit d'une « programmation dans l'interface » [Cypher 1993b]. La programmation conventionnelle demande au programmeur le passage d'une représentation visuelle du déplacement d'un objet à travers l'écran en une structure textuelle totalement différente de ces actions. La programmation dans l'interface autorise l'utilisateur à définir cette opération en réalisant l'action, chose qu'il sait déjà faire. Un tel utilisateur programme et exécute l'action dans le même environnement. Par comparaison, la programmation conventionnelle implique d'apprendre une seconde méthode pour se référer aux objets ainsi qu'aux actions.

Cypher relève les difficultés majeures de la PbD : il s'agit pour lui de trouver comment « inférer les intentions de l'utilisateur » ; comment généraliser les actions de l'utilisateur en programme ? Que veut signifier l'utilisateur lorsqu'il manipule un objet dans l'exemple ? Quel flot de contrôle pour le programme ? Comment identifier les répétitions ou alternatives nécessaires ?

2.1.3.2 La grille d'évaluation

Permettant de mieux comparer les systèmes entre eux, la grille d'évaluation fournie dans [Cypher, Kosbie, & Maulsby 1993] est organisée autour de quatre points :

- Domaine et utilisateurs. Dans quel domaine le système est-il situé ? Quels sont les utilisateurs visés ? Des programmeurs novices ou non ? Des utilisateurs « communs » ?
- Les moyens d'interaction. Comment crée-t-on un programme ? Comment le système guide-t-il le programmeur dans la construction ? Comment exécute-t-on un programme ? Comment le visualise-t-on et le modifie-t-on ?
- L'inférence. Quel est le moteur d'inférence ? Comment infère-t-on les types d'objets et les structures de contrôle ? Utilise-t-on des exemples multiples ou des exemples négatifs ?
- Les connaissances du domaine. Utilise-t-on des connaissances du domaine pour résoudre les ambiguïtés ? Quel type d'information utilise-t-on ? Quelle en est la source ? Quel est le degré de généralité des programmes construits ?

Nous utiliserons cette grille dans la section 2.2.5 pour résumer les caractéristiques des systèmes que nous avons développés.

2.1.3.3 Quelques exemples

De nombreux outils ont montré, dans différents domaines, la validité de cette approche :

- SmallStar [Halbert 1984] pour la programmation iconique,
- Tels [Witten & Mo 1993], Tourmaline [Myers 1991] ou Grammex [Lieberman, Nardi, & Wright 1998] pour la programmation textuelle,
- Peridot [Myers 1988 ; Myers 1993a] ; Garnet [Hashimoto & Myers 1992 ; Myers, et al. 1990] et Lemming [Olsen, Ahlstrom, & Kohlert 1995] pour la programmation d'interfaces,
- Macros by example [Olsen & Dance 1988],
- Dance [Stastko 1991] pour l'animation d'algorithmes,
- Pavlov [Wolber 1996], KidSim/Cocoa [Cypher & Smith 1995 ; Smith & Cypher 1995] et Gamut [McDaniel 1999 ; McDaniel & Myers 1999] pour les jeux ou la simulation,
- WYSIWYC [Wilde 1993], Geometer's Sketchpad [Jackiw & Finzer 1993], ProDeGe+ [Sassin 1994] ou encore Topaz [Myers 1998] dans le cadre des applications de dessin,
- ASM/GeoNode [Van Emmerick 1989a ; Van Emmerick 1989b ; Van Emmerick 1991], et [Loukipoudis 1996] pour les applications de conception technique.

Malgré cette évidente abondance, le concept de « Programming by Example » n'a pas réussi à s'imposer dans les applications professionnelles, où son utilisation demeure très limitée. Cocoa est probablement le seul système qui soit largement diffusé.

2.2 Contribution

Notre contribution au domaine a débuté, dans le cadre d'une thèse de Doctorat [Girard 1992], par la résolution des problèmes posés par le paramétrage d'objets techniques dans les systèmes CAO. C'est ainsi que les systèmes Like [Girard 1992] et EBP [Potier 1995] ont validé les idées successives sur le sujet. GIPSE [Patry 1999a] constitue la troisième étape vers la définition d'un véritable environnement de programmation qui, tout en demeurant dans le domaine de la CAO, effectue une transition vers le domaine traité dans le Chapitre 3.

Cette section est organisée comme suit : tout d'abord (2.2.1), nous présentons notre propre contribution au travail de clarification de Halbert et Myers sur les notions évoquées tout au long de la section 2.1 et nous proposons une terminologie française comme traduction des termes anglais introduits par différents auteurs. Nous proposons également une taxinomie centrée sur la notion de programmation, qui permet à notre sens une meilleure classification des notions que recouvre le domaine du « End-User Programming ». Dans un deuxième temps (section 2.2.2), nous exposons en quoi la CAO constitue un domaine de choix pour le « End-User Programming ».

Ensuite, nous développons deux points particuliers, celui des structures de contrôle (2.2.3) et celui des environnements de programmation (2.2.4). Dans le premier, nous montrons comment nous avons résolu les problèmes liés à l'introduction de structures de contrôle générales dans les systèmes de PbD. Dans le second, nous décrivons les systèmes auxquels nous avons contribué, qui tendent à la définition de véritables environnements de programmation par PbD.

Enfin, nous résumons notre contribution avant d'en envisager les perspectives.

2.2.1 Terminologie et taxinomie : PsE / PbD

Notre contribution au domaine a débuté par une proposition de taxinomie des environnements de programmation. En effet, les diverses classifications proposées, encore partielles, souffraient encore de grandes imprécisions, et ne permettaient pas de caractériser tout un ensemble de travaux dont les nôtres. Nous en avons profité pour définir des traductions françaises.

Après avoir discuté du rôle de l'exemple dans le processus de programmation, nous détaillons la notion d'inférence telle que définie par Myers. Puis, nous proposons des critères totalement orthogonaux pour classer les environnements, définissant une taxinomie que nous utiliserons au fil de toute cette section.

2.2.1.1 Rôle de l'exemple en programmation

Toutes les expressions utilisées par les auteurs incluent le terme « programming », qui correspond au terme français « programmation ». Myers [Myers 1990c] propose d'utiliser les expressions « Programming with Example » et « Programming by Example », et de les regrouper sous l'expression « Example-Based Programming », alors que les travaux ayant abouti à l'ouvrage « Whatch What I Do » introduisent l'expression « Programming by Demonstration ». Quel est le lien entre ces différentes expressions ? Comment peuvent-elles se traduire en français ?

Afin de répondre à ces questions, il est nécessaire de dépasser la simple traduction littérale pour définir précisément la sémantique sous-jacente. Le but des nouvelles approches de la programmation que nous étudions ici est de faciliter la conception de programmes en diminuant le niveau d'abstraction exigé du programmeur. Qu'est-ce qu'un programme ? Quelles sont les principales difficultés d'écriture d'un programme ?

2.2.1.1.1 Programme et algorithme

Avec l'apparition de nouvelles classes de langages de programmation, la notion de programme a progressivement évolué. Si le but final d'un programme est de commander un processeur et donc de décrire, d'une certaine manière, l'algorithme qu'il doit exécuter, la forme de description elle-même peut être assez variable. Partons donc de la notion d'algorithme [Pierra 1991] :

« Étant donné une action abstraite à réaliser et un processeur défini par l'ensemble des [classes d'] objets qu'il sait manipuler et la liste de ses actions primitives, on appelle algorithme l'énoncé de l'ensemble [structuré] d'actions primitives permettant de réaliser cette action, chaque énoncé d'action primitive comportant la désignation des objets, constantes ou variables, sur lesquels elle doit porter. »

Cette définition fait clairement apparaître les deux difficultés majeures de la programmation :

- Un programme porte sur des *variables*, et c'est précisément cette notion de variable qui permettra de décrire en un seul algorithme plusieurs processus.
- L'algorithme est constitué d'un ensemble d'actions primitives ; cependant, cet ensemble d'actions n'est pas, en général, exécuté de façon séquentielle, mais selon une certaine structure *de contrôle de l'exécution.*

2.2.1.1.2 Le rôle de l'exemple

Dans un système interactif usuel, l'utilisateur manipule des *valeurs* (3, le fichier « monfic », « tel arc de cercle que je désigne »). Passer à une notion de programme nécessite d'y substituer, pour certaines valeurs au moins, une notion de *variable* : l'action décrite porte sur le nom de la variable, l'exécution porte sur le contenu. Une même description factorise donc plusieurs exécutions.

Cette notion de variable est la première difficulté de la programmation. Elle exige de l'utilisateur de raisonner sur une abstraction qui n'a pas de valeur unique mais qui représente un ensemble possible de valeurs. C'est cette difficulté que vise à atténuer l'exemple : en associant à toute variable la valeur particulière que prend celle-ci pour l'exemple d'exécution en cours, l'« Example-Based Programming » vise à permettre à l'utilisateur de raisonner autant que possible en termes de valeurs concrètes et non en termes d'ensembles abstraits de valeurs. La traduction de l'expression par *« programmation basée sur exemple »* ou plus simplement *« programmation sur exemple »* semble tout à fait adaptée.

La définition de Myers concernant la *programmation sur exemple* n'est cependant pas suffisamment précise. Ainsi, au fil des articles réactualisés de Myers [Myers 1986 ; Myers 1988 ; Myers 1990c], on peut constater que HI-VISUAL [Yoshimoto, et al. 1986] passe de la catégorie *sans exemple* (VP, not EBP) à la catégorie *avec exemple* (VP, EBP).

Il s'agit d'un système de programmation qui permet de manipuler des icones afin de construire des programmes selon un formalisme de flots de données. La création du programme se fait par sélection d'une icone qui est ensuite « introduite » dans le programme ; cette icone est alors immédiatement activée, le système « répondant » par la création d'une icone symbolisant les données résultant de cette exécution. Celle-ci peut ensuite être connectée à une autre icone action, permettant l'enchaînement d'une nouvelle action. L'utilisateur ne peut interagir avec un exemple d'exécution. Néanmoins, afin de l'aider à vérifier la correction de ce qu'il décrit, un exemple se déroule en parallèle.

En 1988, Myers ne place pas ce système dans les systèmes *avec exemple* car l'utilisateur n'interagit pas avec l'exemple. En 1990, il le fait car un exemple s'exécute néanmoins simultanément.

SmallStar [Halbert 1984] fonctionne de manière sensiblement différente : on ne programme pas, même visuellement, mais on fait un exemple de ce que devrait réaliser le programme. Au fur et à mesure des interactions, un programme est construit.

Ces deux procédés relèvent à notre sens d'une différence essentielle :

- dans le premier cas, le programme décrit porte sur des *variables*. Celles-ci sont explicitement désignées par l'utilisateur qui ne peut pour cela se servir de l'exemple. La programmation n'est pas *sur exemple*. C'est la représentation *a posteriori* du programme – « *Visual Environment* » dans la terminologie de Shu [Shu 1986], « *Program Visualization* » dans la terminologie de Myers [Myers 1986] – qui est associée à un exemple. L'exemple

permet seulement une représentation plus concrète de ce que l'on a écrit et non une description plus concrète de ce que l'on veut.

- Dans le deuxième cas, l'exemple est un outil de construction. La désignation d'une valeur de l'exemple a pour but de permettre au système d'identifier la variable sur laquelle porte le programme. La programmation est effectivement *sur exemple*. Nous proposons pour la qualifier l'épithète « sur exemple ».

Un environnement de construction de programme, ce que nous appellerons un *environnement de programmation*, est dit « *sur exemple* » si l'utilisateur peut, au moins dans certains cas, utiliser les valeurs de l'exemple pour indiquer au système sur quelle(s) variable(s) implicite(s)doit porter une action.

Un outil de visualisation de programme est dit « *avec exemple* » si un exemple d'exécution du programme peut être montré à l'utilisateur à l'issue de certaines étapes de construction.

Tout système de programmation « sur exemple » est évidemment associé à un outil de visualisation de programme « avec exemple » : l'exemple sur lequel le programme est construit s'exécute nécessairement au fur et à mesure que celui-ci est défini.

Au contraire, un outil de visualisation de programme « avec exemple » est complètement indépendant du système de programmation utilisé : il est toujours possible d'associer un tel outil à tout système de programmation. Plusieurs exemples peuvent même s'exécuter en parallèle pour autant que cela ait de l'intérêt pour l'utilisateur. Bien sûr la notion d'outil de visualisation de programme « avec exemple » est surtout intéressante pour les programmes interprétés (Lisp ou BASIC par exemple) ; dans ce cas, l'exécution de l'exemple peut être réalisée instruction par instruction, apportant ainsi une réelle aide au programmeur. La programmation n'en reste pas moins traditionnelle et ne doit pas être qualifiée de « sur exemple ».

2.2.1.2 La notion d'inférence

L'inférence est considérée par les auteurs comme particulièrement importante dans les systèmes de « Programming by Demonstartion » [Cypher 1993b]. Sa définition, qui conditionne l'ensemble des travaux de classification déjà mentionnés, est pourtant particulièrement floue. À partir d'un point de vue purement linguistique, nous démontrons que l'utilisation par les auteurs de cette notion est par trop imprécise. Ceci nous amène ensuite à proposer, à partir d'une clarification des concepts sous-jacents, et par là même, à préciser le vocabulaire.

Myers [Myers 1988] différencie deux catégories de programmation avec exemple : « Programming *with* Example » et « Programming *by* Example ». La traduction littérale des deux expressions ne peut être faite correctement, car nous avons déjà réservé la conjonction « avec » (traduction de « with ») pour une catégorie différente de systèmes. Quant à la conjonction « par », traduction littérale de « by », son acception française est-elle suffisamment claire ? « À travers » n'est-il pas plus indiqué ? La notion de démonstration, elle, n'est pas équivalente en français et en anglais : alors qu'en anglais, elle inclut la notion d'exemple « a sign or an example of something » [Hornby 1995], son acception française en est dépourvue et ne comprend que des aspects déductifs (raisonnement déductif) ou d'exhibition (action de montrer, sous les yeux d'une assistance…) [Robert 1971]. Dans tous les cas, elle recouvre le processus explicatif, ce qui réduirait son champ aux seuls cas « by example » de Myers.

Revenons à l'origine de cette distinction de vocabulaire : Myers base la différence entre ces deux expressions sur la notion d'inférence, qu'il définit comme « l'aptitude du système à générer de nouveaux faits à partir d'autres informations ». La présence d'un tel mécanisme le fait qualifier un système de « *by* Example », alors que ceux qui en sont dépourvus sont qualifiés de « *with* Example » Cette définition s'appuie sur l'acception anglaise du terme « *to infer: to reach an opinion on available information or evidence; to arrive at a conclusion* » [Hornby 1995].

Elle est cependant insuffisamment précise car, ne distinguant pas l'utilisateur et le système, elle ne spécifie pas si les « nouveaux faits » sont nouveaux par rapport *aux faits déjà connus* de l'utilisateur,

ou si les « nouveaux faits » le sont uniquement par rapport aux *seules informations transmises* par l'utilisateur. Ceci aboutit à confondre deux problèmes différents:

- Quelles inférences sont possibles au niveau du système pour éviter à l'utilisateur tout ou partie de la démarche de conception de l'algorithme ?
- Comment permettre à l'utilisateur de transmettre au système, de façon précise et commode, l'ensemble des faits qu'il connaît déjà sur l'algorithme ?

Ainsi, les deux systèmes suivants sont considérés par Myers comme appartenant à la même classe de systèmes inférentiels (dite « Programming by example ») :

· Le système décrit par Shaw [Shaw 1975] permet d'inférer la structure d'un programme Lisp de traitement de chaînes de caractères à partir d'un exemple (simple !) de couples de chaînes en entrée et en sortie, associé, durant la phase d'inférence à un dialogue avec l'utilisateur :

```
(A B C D) => (DDCCBBAA)
```

· Dans le système décrit par Bauer [Bauer 1979], pour distinguer constantes et variables, l'utilisateur décrit deux exemples. S'il change la valeur d'une certaine donnée entre les deux exemples, le système « infère » qu'il s'agit d'une variable. Si une même donnée possède deux fois la même valeur, le système « infère » qu'il s'agit d'une constante :

```
TO FIND-ROOT OF 5 DO
5 IS GREATER-THAN 0 SO
TO FIND-ROOT OF – 1 DO :
       LET R BE 1.
– 1 IS NOT GREATER-THAN 0 SO
       LET S BE R PLUS 1.
       THE ANSWER IS 0.
S2 IS NOT GREATER-THAN 5 SO
       LET R BE S.
ADD 1 TO S.
LET S2 BE S TIMES S.
S2 IS GREATER-THAN 5 SO
THE ANSWER IS R.
```

Le « type d'inférence » utilisé dans ces deux systèmes est fondamentalement différent :

1. Dans le premier cas, le système essaie d'inférer un algorithme de résolution que l'utilisateur ne connaît pas ou ne souhaite pas décrire explicitement à partir d'un simple exemple de résultat. Le système doit réellement inférer, c'est-à-dire générer de nouveaux faits (le programme général) à partir de faits déjà connus (l'exemple).
2. Dans l'autre cas, l'utilisateur sait parfaitement quelles sont les constantes et les variables. Il souhaite seulement faire connaître commodément cette information au système. Aucune inférence, aucune génération de nouveaux faits, à partir de faits déjà connus de l'utilisateur, n'est nécessaire pour le système : le seul problème est de définir une convention de dialogue, précise et « naturelle » pour l'utilisateur, lui permettant de transmettre au système un fait parfaitement connu de lui.

Cette confusion entre inférence et convention de dialogue amène même Myers à considérer [Myers 1990c] que le programme MacDraw® (CLARIS) doit être classé dans les systèmes à inférence parce que la commande « Duplicate » utilise une transformation modale. Le fait que le système mémorise, à l'occasion d'une duplication, la transformation utilisée, et qu'il réutilise, par défaut, cette même transformation lors des duplications suivantes, est considéré comme une inférence, alors qu'il s'agit d'une convention de dialogue ; ceci implique que cette convention doit être clairement connue de l'utilisateur pour lui permettre d'utiliser le système. Lieberman [Lieberman 1990] écrivait déjà en 1990 qu'il ne fallait pas confondre l'approche utilisée dans Tinker (assimilable au cas 2) avec ce qu'il appelle « programmation par l'exemple de l'histoire des entrées-sorties » (assimilable au cas 1). Notons également l'avant-dernière phrase de l'introduction de Cypher définissant le domaine de la « programming by Demonstration » : « … *the success of a PbD system depends far more of the user experience of interacting with the system than it does on the induction algorithms used to create the users' programs.* ».

Compte tenu de toutes ces difficultés, nous avons choisi de pas utiliser le terme « d'inférence » mais d'introduire une distinction plus simple entre approche déclarative et impérative.

2.2.1.2.1 Notion d'inférence et structure d'exécution : programmation déclarative et impérative

Outre la difficulté liée à la notion de variable, réduite par l'introduction de l'exemple, nous avons vu que la deuxième grande difficulté de conception d'un algorithme est l'élaboration de la structure de contrôle de son exécution. Dans de nombreux cas, en particulier dans les systèmes interactifs de modélisation, l'utilisateur est capable d'identifier les actions primitives à réaliser (il sait en exécuter un exemple). Il lui est en revanche très difficile, en particulier si des structures alternatives ou répétitives interviennent, de définir de façon abstraite la structure de contrôle de l'exécution de ces primitives pour les différents cas d'exécution du programme.

En programmation traditionnelle, une classe particulière de langages de programmation a même été développée pour tenter d'éviter cette difficulté : c'est la classe des langages de programmation déclarative. Dans un langage déclaratif, le programmeur ne décrit pas la structure de contrôle d'exécution, le séquencement des actions primitives qui constituent l'algorithme : celui-ci (souvent appelé le « moteur ») est préexistant. L'utilisateur lui fournit seulement des informations sur les *objets* qu'il doit traiter et les *relations entre objets* qu'il doit prendre en compte lors de son exécution. Les relations entre objets peuvent être très diversement définies, par exemple sous forme de règles (Prolog II), sous forme de contraintes (PROLOG III, programmation géométrique par contraintes [Dufourd 1990 ; Verroust 1990]), ou encore sous forme d'expressions arithmétiques (les tableurs tels Excel®, Lotus 1-2-3®, ...). Ces relations sont alors exploitées pour fournir à partir de données d'entrée les données de sortie du programme.

Les difficultés concernant les relations entre objets sont de nature très différente de celles concernant la définition du séquencement propre à la programmation impérative, et sont souvent considérées comme moindres. Les relations peuvent, en général, être décrites dans un ordre quelconque. La seule difficulté pour l'utilisateur est de s'assurer que l'ensemble des relations définies est à la fois non contradictoire et suffisant pour permettre au moteur de trouver le résultat souhaité pour tout ensemble de données d'entrées.

Remarquons que les difficultés de programmation concernant les objets traités sont tout à fait analogues en programmation déclarative et impérative (notion de variable, distinction variable/constante, distinction des paramètres et des variables internes). L'analyse que nous avons effectuée sur le rôle possible d'un exemple pour limiter cette difficulté s'applique donc également à la programmation déclarative : il est plus facile de décrire les relations entre deux variables sur des valeurs particulières de ces variables, correspondant à un exemple, que sur les variables abstraites. Le fait d'utiliser ou non un exemple apparaît donc orthogonal à la classification impératif/déclaratif.

Nous proposons d'utiliser le critère déclaratif/impératif plutôt que le terme d'inférence : pour être qualifié de déclaratif, le système doit réellement déduire des faits nouveaux (comment exploiter les relations définies par l'utilisateur) à partir des seuls faits connus de l'utilisateur (les objets et relations entre objets). Dans tous les autres cas, il s'agit de programmation impérative sur exemple. De fait, ceci correspond à une définition plus restrictive de l'expression de Myers « Programming *by* Example »

2.2.1.2.2 Notion d'inférence et interface homme-machine

Concevoir un programme, et, pour le cas qui nous occupe, passer d'un exemple particulier à un programme général, nécessite, dans tous les cas, que certaines informations complémentaires soient connues de l'utilisateur et puissent lui être communiquées. Si le chiffre « 3 » par exemple est introduit par l'utilisateur dans un exemple d'exécution, il est impossible pour le système de savoir si ce chiffre doit être considéré comme une constante et rangé en tant que tel dans le programme, ou s'il est un paramètre qui doit être redemandé à l'utilisateur lors de chaque exécution. Aucun mécanisme d'inférence n'est ici possible ni souhaitable. Il s'agit de choisir et de spécifier une convention de dialogue. Comme dans tout système interactif [Newman 1979] cette convention doit bien entendu être clairement connue et comprise par l'utilisateur de sorte qu'il puisse l'utiliser pour construire son programme. Ainsi la rémanence de certaines commandes de MacDraw® n'a rien à voir avec de l'inférence : c'est un choix particulier de convention de dialogue qu'il convient d'évaluer

et de discuter en terme d'ergonomie. De même, dans le système « Programming by Examples » [Bauer 1979], qui est un système impératif, le fait de devoir attribuer deux valeurs différentes à la même donnée dans deux exemples différents, pour spécifier qu'il s'agit d'une variable et non d'une constante, est une convention de dialogue. Elle doit être explicitement connue de l'utilisateur, et analysée en terme d'ergonomie. Dans le système Peridot décrit par Myers [Myers 1988], qui possède quelques caractéristiques de programmation déclarative, et utilise donc une forme d'inférence pour générer le séquencement d'exécution, le fait d'interpréter une chaîne de caractère, présentée dans l'exemple « cadrée à gauche », comme devant être cadrée à gauche dans toute exécution, ne relève pas de l'inférence. Il relève d'une convention de dialogue qui doit être explicite et connue de l'utilisateur.

Nous proposons de qualifier les systèmes basés sur exemple dans lesquels l'utilisateur définit explicitement l'ordre d'exécution des primitives de systèmes *impératifs sur exemple*. Nous appellerons *système déclaratifs sur exemple* les systèmes dans lesquels le système doit lui-même générer l'ordre d'exécution de certaines relations ou primitives lors d'une ré-exécution du programme.

2.2.1.3 Taxinomie des systèmes de programmation

Nous avons analysé, au cours des sections précédentes, certaines insuffisances des critères de classification qui avaient été proposés par Myers pour les différents systèmes de programmation. Cette analyse nous a amené à revoir cette classification, et à proposer de classifier les systèmes de programmation selon quatre critères complètement orthogonaux.

2.2.1.3.1 Compilé / interprété

La définition des langages compilés/interprétés est relativement floue, comme l'indique Myers, car il n'existe pas de réelle discontinuité. Nous distinguerons sous le vocable de systèmes à *compilation* ceux qui effectuent la nécessaire traduction (prélude à l'exécution) de façon globale, des systèmes que nous qualifierons *d'interprétés* car effectuant cette même traduction de façon partielle, alors que le processus de conception du programme n'est pas terminé.

2.2.1.3.2 Programmation graphique ou non

Un système de programmation est dit *graphique* (en anglais « Visual Programming ») si la définition du programme peut être effectuée par interaction graphique dans un espace à deux dimensions au moins.

2.2.1.3.3 Programmation sur exemple ou non

Un système de programmation est dit *sur exemple* si l'utilisateur peut utiliser les valeurs d'un exemple d'exécution pour définir les objets sur lesquels porte le programme à construire.

2.2.1.3.4 Déclaratif ou impératif

Un système de programmation est dit *impératif* si la structure de contrôle de l'exécution de l'algorithme résultant du programme est décrite par le programmeur. Il est dit *déclaratif* si la structure de contrôle d'exécution est définie par le système à partir de relations entre objets définies par le programmeur.

Les deux formes les plus modernes de programmation par un non-programmeur sont respectivement les systèmes graphiques impératifs sur exemple et les systèmes graphiques déclaratifs sur exemple. La première forme est pratiquement toujours associée à une notion d'interprétation car l'interpréteur est très voisin du système qui permet de construire l'exemple. La deuxième forme est en général associée à une notion de compilation car l'exécution du programme demande un traitement global.

Ainsi que nous le verrons ci-dessous, la plupart de nos travaux se sont situés dans la programmation impérative sur exemple.

2.2.1.4 Illustration de la classification

Le tableau que nous présentons ci-dessous (Tableau 1) se veut simplement une illustration de la taxinomie que nous avons élaborée et présentée dans cette section ; il inclut des systèmes qui ont été mentionnés dans les pages précédentes, et ne prétend aucunement au classement de tous les systèmes et environnements de programmation.

		SUR EXEMPLE		SANS EXEMPLE	
		Textuel	**Graphique**	**Textuel**	**Graphique**
IMPÉRATIF	**Compilé**	Programming by example *[Bauer 1979]*		Ada, C, FORTRAN	Problox *[Glinert 1987]*
	Interprété	Tinker *[Lieberman 1982]*	SmallStar *[Halbert 1984]*	APL, Basic, LISP	LABVIEW
DÉCLARATIF	**Compilé**	Inferring Lisp Program by example *[Shaw 1975]*		PROLOG III	ThinglaB *[Borning 1981]*
	Interprété	Editing by examples *[Nix 1985]*	Peridot *[Myers 1988]*	PROLOG II	Juno *[Nelson 1985]*

Tableau 1 : Illustration de la taxinomie

Deux cases (grisées) apparaissent vides dans ce tableau. Cependant, certains systèmes décrits dans la suite de cette section seront à classer dans cette catégorie, comme les systèmes dits variationnels (2.2.2.2) qui appartiennent à la catégorie des systèmes de programmation graphique, sur exemple, compilée, et déclarative.

Au-delà de cette taxinomie, qui permet d'identifier clairement le rôle et l'utilisation de l'exemple, il apparaît important de bien préciser le vocabulaire utilisé et de spécifier nos choix par rapport à ces définitions.

L'ensemble des travaux que nous décrivons dans cette section 2.2 appartient au domaine de la *Programmation sur exemple*, pendant exact de la « Programming by Demonstration » des auteurs anglo-saxons. C'est pourquoi nous utiliserons désormais le sigle **PsE/PbD** en référence à ce domaine. Pour des raisons que nous développerons dans la section suivante, nous nous sommes limité à des aspects purement impératifs, même si l'utilisation de la terminologie de Myers pouvait amener à utiliser le terme « *d'inférence* » au sens des auteurs anglo-saxons. Mais il s'agit pour nous de ce que nous considérons comme des conventions de dialogue.

2.2.2 La CAO, un domaine de choix pour la PsE/PbD

Dans son livre, *A small matter of programming, perspectives on end-user computing* [Nardi 1993], B. Nardi identifie la CAO comme un candidat naturel à l'application des techniques de PsE/PbD. Cette affirmation est très théorique, car Nardi ne donne aucun exemple de réalisation dans le domaine. Elle se contente de relever de nombreux points où la PsE/PbD serait susceptible de se révéler fort utile. Nous verrons en fait dans cette section que les systèmes dits paramétriques ont déjà permis de franchir un premier pas en ce sens.

D'un point de vue très général, on peut dire que la PsE/PbD est plus naturelle lorsque les objets de l'exemple ont une représentation naturellement graphique. Tinker [Lieberman 1993], par exemple, permet de faire de la PsE/PbD de façon totalement textuelle. Cependant, l'identification de la nature des objets y est délicate, et nécessite l'acquisition par l'utilisateur de concepts tels que « constante » ou « variable ». Même si la représentation symbolique d'objets tels que des icones permet de concrétiser les valeurs de l'exemple, cela n'en demeure pas moins conventionnel. En revanche, le domaine purement graphique, où la manière la plus naturelle de « désigner » un objet est de le « montrer » sur la surface de visualisation, est celui où les avantages liés à la manipulation de l'exemple en lieu et place de leur abstraction sont les plus nets.

2.2.2.1 Les raisons du succès

La première tient à la nature de l'activité de conception technique, la seconde au marché des composants standard.

L'activité de conception technique (qu'elle soit mécanique, architecturale ou autre) suppose l'application de règles strictes dépendant du domaine. En cela, elle n'a rien à voir avec l'activité beaucoup plus libre de conception « artistique ». Au cours du processus de conception, le dessinateur technique (le « concepteur ») connaît parfaitement les relations qui doivent exister entre les différents éléments de son modèle en cours d'élaboration. De ce fait, les systèmes CAO lui permettent d'exprimer explicitement ces relations, généralement sous la forme de contraintes ou d'opérateurs géométriques, et les utilisateurs sont habitués à exprimer ces relations. Le simple fait d'enregistrer ces contraintes fournit une base pour la construction automatique de programmes.

La seconde raison se déduit de l'observation suivante : concevoir un nouveau produit consiste souvent à assembler des composants préexistants. Ces composants, appelés « composants standard » ou « standard parts », sont regroupés en familles décrites par un modèle unique, qui exprime des variations possibles en dimensions, tolérances ou formes [Shah & Mäntylä 1995]. Ces composants peuvent être décrits dans des normes (par exemple, la famille de vis hexagonales ISO 1014), ou bien être propres à un fournisseur ou à une entreprise utilisatrice. Dans la première génération de systèmes CAO, ces familles étaient décrites sous la forme de programmes paramétrés, en langages généralistes (FORTRAN) ou spécialisés. L'inconvénient majeur de cette approche réside dans le fait qu'une double compétence (dessin technique / informatique) est alors requise. Écrire de véritables programmes n'est pas à la portée de tous les utilisateurs, même dessinateurs. La PsE/PbD offre alors une solution séduisante pour permettre la production de ces programmes par des dessinateurs.

2.2.2.2 Les systèmes variationnels et paramétriques

Alors que tous les systèmes modernes de CAO offrent la possibilité d'exprimer les contraintes entre objets lors de la création de ces derniers, l'enregistrement de ces contraintes dans le modèle n'est en fait apparu que récemment. Bien que le système MEDUSA [Newell, Parden, & Parden 1983] puisse être considéré comme le premier système ayant fourni dès 1983 la possibilité d'enregistrer des contraintes, la généralisation de cette possibilité ne s'est faite qu'à la fin des années 1980. Une nouvelle génération de systèmes est alors apparue, capable de régénérer un nouveau modèle après modification d'une ou plusieurs contraintes enregistrées dans un modèle donné. En fait, ces systèmes, appelés « Dimension Driven Systems » (DDS) [Roller 1990], ont une structure interne et un comportement doubles. Tout d'abord, ce sont des systèmes classiques, qui permettent à l'utilisateur de créer un modèle, de le visualiser, et de le modifier. Mais, de plus, l'utilisateur peut généraliser son modèle (et ainsi créer un programme) en demandant l'enregistrement des contraintes et des valeurs qui sont impliquées dans celui-ci ; la modification des valeurs permet au système de générer automatiquement un nouveau modèle, correspondant à l'application des mêmes contraintes sur ces nouvelles valeurs. En fait, ces systèmes peuvent être divisés en deux catégories, qui correspondent à la taxinomie présentée préalablement : les approches déclarative et impérative.

Les **systèmes variationnels** cachent un **programme déclaratif**. L'utilisateur construit un exemple, et spécifie les contraintes, implicitement ou explicitement. Le système enregistre ces contraintes sous forme d'un ensemble d'équations, dont un solveur peut tirer une solution. La méthode de création de l'objet peut demeurer inconnue du dessinateur. De nombreuses méthodes ont été utilisées pour résoudre ces systèmes de contraintes, les plus efficaces semblant être aujourd'hui basées sur la réduction de graphes [Bouma, et al. 1995 ; Owen 1991]. Malgré de réels progrès, cette approche souffre de certaines limites : (1) L'ensemble d'équations a généralement plus d'une solution, et seules des heuristiques ont pu être définies pour capturer « l'intention » de l'utilisateur. Des exemples simples ont été publiés [Bouma et al. 1995] qui ont démontré le côté inattendu des solutions parfois trouvées. (2) À cause de la nature « heuristique » de ces solutions, deux systèmes différents peuvent ne pas trouver la même solution. (3) L'approche variationnelle pure n'est pas applicable à beaucoup d'opérations, principalement en 3D. C'est pourquoi tous les

systèmes dits variationnels sont en fait hybrides, et utilisent une approche variationnelle en 2D et l'approche impérative en 3D.

Les **systèmes fonctionnels**, souvent appelés **systèmes paramétriques**, cherchent à résoudre un problème très différent : étant donné une classe de composants dont le processus de conception est bien connu, on veut que chaque instance de la classe puisse être générée automatiquement à partir de la valeur des paramètres qui la caractérisent. Ces systèmes cachent une composition de fonctions analogue à un **programme impératif**. Ce programme est souvent capturé au cours de l'exécution du système interactif : le système CAO « espionne » le dessinateur au fur et à mesure que ce dernier construit son exemple. Par la suite, le système CAO est à même de rejouer la séquence de construction, avec éventuellement de nouvelles valeurs. La représentation interne des programmes peut être textuelle, mais elle est plus généralement basée sur des structures de données [Pierra, Potier, & Girard 1994 ; Solano & Brunet 1994] telles que les graphes orientés acycliques [Cugini, Folini, & Vicini 1988]. Le système Pro-Engineer®[1] est l'exemple le plus populaire de cette approche. Notons que rien n'interdit d'éditer les fonctions de construction, voire de les modifier. Les systèmes de type fonctionnel ont donc un domaine d'application beaucoup plus large que la simple ré-exécution d'un historique.

Dans le domaine de la CAO, l'approche DDS est si séduisante qu'aujourd'hui, tous les systèmes de CAO se doivent de posséder de telles possibilités. Cette diffusion très large démontre l'intérêt pratique de l'approche. Elle démontre également que les dessinateurs, utilisateurs finaux, sont capables de générer des formes paramétrées, autrement dit de réels programmes visuels, sans notion de programmation. Cela ne signifie cependant pas absence de modification du processus de conception. En effet, dessiner un modèle et dessiner une famille de modèles sont deux pratiques sensiblement différentes. Cependant, cette activité ne se situe pas au niveau d'abstraction de la programmation classique, et les systèmes paramétriques réduisent considérablement l'effort nécessaire pour la conception de familles de composants.

2.2.2.3 Un choix primordial : pas d'inférence

La distinction entre les systèmes paramétriques et variationnels correspond exactement à la distinction entre systèmes impératifs et systèmes déclaratifs que nous avons proposée à la section 2.2.1.3. Les premiers sont des systèmes de programmation graphique impératifs sur exemple, alors que les autres sont des systèmes de programmation graphique déclaratifs sur exemple. En revanche, la distinction entre systèmes « compilés » et « interprétés » est fonction des systèmes étudiés. Initialement, les systèmes variationnels devaient être considérés comme compilés selon notre définition, car l'évaluation du « programme » (l'ensemble des contraintes définies par l'utilisateur) se faisait globalement. À l'inverse, les systèmes paramétriques permettaient l'enregistrement et la résolution des contraintes de construction progressivement. Aujourd'hui, cependant, les systèmes de CAO proposent souvent des mécanismes combinés, qui relèvent des deux approches.

Pour notre part, nous nous sommes attaché à n'utiliser que des mécanismes algorithmiques ne relevant pas de l'inférence. La raison majeure tient au fait que les utilisateurs des systèmes CAO sont habitués à exprimer leurs choix, et nous avons toujours privilégié une solution transparente pour eux. Cependant, les choix que nous avons effectués entre les différentes possibilités de conventions de dialogues peuvent être modifiés en fonction des évaluations des utilisateurs sans remettre en cause la nature des solutions que nous avons proposées.

2.2.3 Structures de contrôle

Parmi les problèmes relevés par Myers et Cypher comme les plus importants dans le domaine de la PsE/PbD, celui de la définition interactive des structures de contrôle était l'un des plus cruciaux. En 1990 [Myers 1990c], l'absence (ou tout au moins le caractère très partiel) des structures de contrôle dans les systèmes de PsE/PbD constituait même l'un des obstacles majeurs à l'utilisation de ce mode de programmation.

1 Parametric Technology Inc. USA.

Par structure de contrôle, il faut entendre principalement les alternatives et les itérations d'une part, et la notion de sous-programme d'autre part.

2.2.3.1 Structures de contrôle et PsE/PbD

Les structures de contrôle supportées par les systèmes de PsE/PbD sont généralement limitées. De nombreux travaux ont cependant visé à introduire de telles structures, et nombre de systèmes en proposent aujourd'hui des versions plus ou moins complètes.

Ainsi, parmi les systèmes qui définissent l'ensemble de leurs programmes par l'exemple, les sous-programmes avec paramètres sont-ils présents dans Chimera [Kurlander 1993a ; Kurlander 1993b ; Kurlander & Feiner 1992], Geometer's Sketchpad [Jackiw & Finzer 1993] et Aide [Piernot & Yvon 1993], mais avec seulement des paramètres d'entrée (ce sont davantage des « macros » que de réels sous-programmes). La récursivité n'est explicitement fournie que dans Geometer's Sketchpad, mais l'absence d'expressions de contrôle dans ce système rend la définition récursive incomplète : l'utilisateur est obligé de préciser à chaque exécution le niveau de profondeur de récursivité qu'il souhaite voir appliquer. Les conditions sont assez souvent présentes (Peridot [Myers 1993a], Turvy [Maulsby 1993], Metamouse [Maulsby, et al. 1992 ; Maulsby, Witten, & Kittlitz 1989], ou Tels [Witten & Mo 1993]), mais se limitent toujours à des traitements de cas très particuliers (existence ou non d'un objet, satisfaction d'une contrainte géométrique). Enfin, les structures répétitives sont des itérations de collections (*set iteration* [Halbert 1984], consistant à appliquer une même action à des objets différents pris dans un ensemble (Peridot) ou dans un texte (Tels). Certains systèmes permettent l'identification automatique de séquences d'interactions répétées (Eager [Cypher 1991 ; Cypher 1993a]), Turvy , Chimera), ou encore permettent l'instanciation de structures répétitives prédéfinies comme la création d'un nombre prédéfini d'objets (Metamouse, Peridot). Peridot est le seul système qui introduit la notion de relations de récurrence, mais ces relations sont déterminées par identification avec des classes prédéfinies.

Enfin, les systèmes considérés comme les plus complets abandonnent le principe de base de la PsE/PbD en demandant à l'utilisateur de modifier textuellement le programme (hors exemple), afin d'introduire les structures de contrôle (SmallStar [Halbert 1984], GeoNode [Van Emmerick 1991], Tinker).

Une partie importante de notre apport dans le domaine a précisément consisté à introduire des mécanismes généraux permettant à un système de PsE/PbD de posséder toutes les structures de contrôle de la programmation impérative, et ce sans demander une quelconque intervention textuelle dans le programme construit. Pour cela, nous avons caractérisé la notion de contexte (2.2.3.2), qui permet d'effectuer la transition entre les objets de l'exemple et les variables du programme en cours de génération. Ensuite (2.2.3.3), nous avons établi les règles de gestion de ce contexte dans le cadre de l'utilisation des structures de contrôle. Enfin (2.2.3.4), nous avons étudié la définition purement interactive de ces structures de contrôle.

2.2.3.2 L'outil de base : le contexte

La principale différence entre programmation traditionnelle et PsE/PbD réside dans le fait suivant : alors que le programmeur traditionnel *décrit* des actions à effectuer sur des objets théoriques (les variables), le programmeur par démonstration *réalise* des actions sur des objets concrets possédant des valeurs. Alors que le premier effectue trois étapes (édition/compilation/exécution), le second alterne simplement deux phases, la phase d'**enregistrement**, au cours de laquelle le programme est enregistré, et la phase d'**exécution**. La tâche d'un système de PsE/PbD comporte essentiellement deux points :

- enregistrer ou ré-exécuter les actions sélectionnées par l'utilisateur,
- substituer les valeurs d'exemple par des variables (enregistrement) et inversement (ré-exécution).

Si le premier point ne présente pas de difficulté particulière tant que l'on demeure dans l'enregistrement séquentiel, le second nécessite une analyse assez fine des données impliquées dans l'exemple.

2.2.3.2.1 Distinction constantes / paramètres

Généraliser un exemple [Cypher, Kosbie, & Maulsby 1993] revient en premier lieu à identifier parmi les objets introduits par l'utilisateur ceux qui demeureront constants pour chacune des exécutions futures, et ceux qui peuvent/doivent varier. Les premiers sont des *constantes*, alors que les seconds constituent les *paramètres* ou *arguments* du programme final. Les constantes doivent être enregistrées en l'état au sein du programme, alors que le nom des paramètres doit se substituer à leur valeur dans le programme enregistré. Lors de l'exécution du programme, les valeurs des constantes sont utilisées telles quelles, alors que de nouvelles valeurs pourront être affectées aux paramètres, valeurs sur lesquelles porteront les actions enregistrées.

Cette distinction n'est cependant pas suffisante. En effet, la programmation traditionnelle ne distingue pas constantes et paramètres (ou arguments), mais plutôt constantes et variables, ces dernières pouvant elles-mêmes être soit des paramètres, soit des variables internes. Or une particularité des systèmes graphiques de conception vient s'ajouter : le principal but d'un processus de conception consiste à *créer* de nouveaux objets à partir des premiers.

2.2.3.2.2 Variables implicites

Du point de vue de la PsE/PbD, ceci revient à construire de nouveaux objets (par les commandes de création disponibles au niveau du système) à partir d'objets créés précédemment. Traduit en terme de programmation, il s'agit de *déclarer* de nouvelles variables (création) puis de leur *affecter* pour valeur dans l'exemple le résultat de l'action de création. Il faut alors, dans le programme, effectuer une substitution valeur/nom identique à celle décrite ci-dessus pour les paramètres. Nous avons appelé ces objets des *variables implicites*.

Une difficulté majeure se présente alors : celle de l'identification des variables. Si, pour les paramètres, qui sont dans la pratique en nombre limité, demander à l'utilisateur de préciser leur nom paraît raisonnable (il pourra en particulier leur donner un nom significatif), agir de même pour les objets créés durant l'enregistrement d'un programme entraînerait la disparition de bien des avantages de la PsE/PbD. Dans la quasi-totalité des cas (hors structures de contrôle), le dessein de l'utilisateur est pourtant clairement identifiable : il s'agit d'une référence temporelle, c'est-à-dire de la référence au énième objet créé pendant l'exemple en cours. Une simple numérotation systématique des objets créés durant la définition de l'exemple permet ainsi d'identifier chacune des variables, et d'effectuer automatiquement les substitutions requises. Cette approche apparaît suffisante pour des programmes purement séquentiels.

2.2.3.3 Contexte et structures de contrôle

Les structures de contrôle (au sens large) présentent une incompatibilité majeure avec la gestion du contexte implicite que nous avons proposée ci-dessus. En effet, une exécution correcte, selon les règles précédentes, suppose une substitution nom/valeur en phase d'exécution équivalente à la substitution valeur/nom de la phase d'enregistrement. Cette équivalence repose implicitement sur le fait que les objets sont temporellement créés dans le même ordre. Le quatrième objet créé lors de l'enregistrement du programme est supposé être le même objet lors de toute exécution. Ceci est vrai pour tout enregistrement purement séquentiel, en l'absence d'échec d'une commande.

Mais dans le cas de l'introduction des structures de contrôle, il n'en est pas de même. Quel sens peut avoir la désignation d'un objet durant la branche ALORS d'une structure alternative si le programme est passé lors de l'exécution par la branche SINON ? On pourrait envisager de le substituer par l'objet de rang équivalent créé dans la branche SINON, mais existe-t-il ? La numérotation automatique des variables est basée sur le principe d'un nombre constant de créations d'objets à chaque exécution. La création d'un nombre différent d'objets entre les deux branches d'une alternative créerait ainsi un décalage dans la numérotation des variables créées après l'alternative.

Ce problème est encore plus crucial dès lors qu'une structure répétitive est créée. Sauf cas particulier, une structure répétitive réalise un nombre d'itérations différent à chaque exécution. La

numérotation ultérieure des variables est donc naturellement affectée. Quant à la désignation d'objets construits au cours de l'itération, quel sens a-t-elle ? Que signifie la désignation d'un objet construit au cours du troisième tour d'une itération en comptant quatre lors de la phase d'enregistrement, lorsque l'exécution de la même structure compte sept itérations ? S'agit-il d'un objet du troisième tour (constante temporelle stricte) ? ou du sixième tour (avant-dernier tour) ? Pis encore, qu'en est-il dans le cas où la structure ne comprend que deux tours ?

Nous verrons ci-dessous comment le principe général des sous-programmes peut être appliqué aux autres structures pour fournir à ces problèmes une solution cohérente.

2.2.3.3.1 Sous-Programmes

Telle que les langages modernes (tels PASCAL ou Ada) la réalisent, la gestion des sous-programmes s'adapte parfaitement aux besoins de la PsE/PbD. Les paramètres du sous-programme constituent l'interface d'échange avec le programme appelant. Les autres variables du sous-programme représentent des variables locales à ce dernier, de l'extérieur duquel elles sont inaccessibles. La réentrance du code suppose, en revanche, que ces mêmes variables locales soient créées à nouveau à chaque appel.

En fait, cette assimilation est un peu rapide : assimiler les seuls paramètres, tels que nous les avons définis dans la section précédente, à l'interface d'un sous-programme est très réducteur. En effet, seuls des paramètres d'entrée ont, pour l'instant, été introduits. Or, les langages classiques proposent également des paramètres de sortie, ou d'entrée-sortie, dont la définition permet au programme appelant d'exploiter les résultats du sous-programme. Dans le cas de la PsE/PbD, une solution naturelle peut être envisagée : les variables internes du sous-programme, essentiellement graphiques, représentent en fait **globalement** le paramètre de sortie implicite du sous-programme. Elles constituent un ensemble auquel l'utilisateur pourra ensuite accéder, par exemple pour globalement leur faire subir une transformation géométrique ou une duplication. Ces différentes observations nous conduisent aux axiomes suivants, qui gouvernent la gestion des sous-programmes dans Like [Girard 1992] :

- Tout programme peut être considéré comme un sous-programme ; ses paramètres constituent les paramètres d'entrée du sous-programme.
- L'ensemble des objets créés au cours d'un sous-programme (contexte implicite) est intégré comme un tout dans le contexte implicite du programme appelant. Il ne pourra ensuite être manipulé dans celui-ci que globalement.
- À chaque exécution du sous-programme, un nouveau contexte est créé (réentrance).

Ces axiomes, implémentés dans Like, ont prouvé, malgré leur caractère restrictif en ce qui concerne les paramètres de sortie, leur utilisabilité. Cependant, l'axiome suivant a été ajouté dans le système EBP [Potier 1995] :

- Dans un programme PsE/PbD, des paramètres de sortie peuvent être explicitement définis, en nombre obligatoirement fixe, et insérés dans le contexte du programme appelant, permettant une manipulation individuelle dans ce dernier.

Notons que, lorsqu'un sous-programme est appelé à partir d'un autre (sous-) programme, ses paramètres effectifs sont introduits par l'intermédiaire de la calculette grapho-numérique, sous la forme d'une expression appartenant au contexte englobant. Il s'agit donc d'un réel mécanisme de sous-programme. Les axiomes de gestion du contexte que nous avons données ci-dessus permettent d'établir un cadre de réalisation pour toutes les structures de contrôle, comme nous allons le détailler dans la section suivante.

2.2.3.3.2 Alternatives, répétitions et contexte

Les alternatives et les répétitions peuvent être globalement considérées comme des sous-programmes simplifiés. Comme ces derniers, leurs variables implicites constituent un ensemble d'objets dont la manipulation individuelle n'a pas vraiment de sens, mais dont la manipulation globale peut/doit être autorisée.

À chaque structure, doit donc être associé un contexte propre (contexte implicite). Toutes les actions internes à la structure peuvent se référer non seulement aux variables de ce contexte, mais également à celles du contexte englobant (celui de la séquence dans laquelle est définie la structure de contrôle). Chacune de ces références externes définit implicitement un « paramètre d'entrée ». La gestion du contexte dynamique d'un programme (en cours d'exécution) sous la forme d'une pile de contextes, avec insertion globale de chacun dans son contexte englobant, en fin d'exécution, permet ainsi de répondre aisément aux règles d'accessibilité des variables.

Ainsi, une alternative simple (SI-ALORS-SINON) aura-t-elle un contexte unique, constitué de deux sous-contextes, l'un pour la branche ALORS, l'autre pour la branche SINON. L'expression de contrôle de l'alternative pourra se référer exclusivement aux variables du contexte englobant. Pour une répétition, le contexte sera divisé en sous-contextes, un pour chaque tour de la répétition. Le contexte du tour courant sera accessible à l'expression de contrôle de la répétition. Enfin, pour permettre l'imbrication des structures répétitives, tout comme la réentrance pour les sous-programmes, les contextes implicites des alternatives et des répétitions sont créés au début de chaque exécution, puis fermés lors de l'achèvement de chaque structure, et inclus dans le contexte englobant.

2.2.3.3.3 Récursivité

Compte tenu des principes que nous avons décrits ci-dessus, la prise en compte de la récursivité résulte simplement de l'application des axiomes définis (principalement ceux concernant la réentrance), et de la définition de l'alternative qui termine la récursivité.

2.2.3.4 Définition interactive des structures de contrôle

Enregistrer par démonstration un programme séquentiel est une activité simple, qui nécessite seulement l'identification des paramètres. Lorsque l'environnement de programmation, comme c'est le cas pour Like (premier système développé [Girard 1992]), permet la mise au point et la modification des programmes enregistrés, il faut de plus assurer une gestion fine du contexte, pour ajuster correctement la numérotation automatique des variables, lorsque l'utilisateur insère ou supprime une opération constructive. Il faut vérifier la cohérence du programme résultant de la modification. Ainsi, lorsque l'on supprime une action générant un objet, toutes les actions ultérieures utilisant cet objet deviennent incohérentes.

Permettre à l'utilisateur de construire interactivement des structures de contrôle présente des difficultés supplémentaires, tant au plan du système de PsE/PbD que de celui des conventions de dialogue à définir.

2.2.3.4.1 Sous-Programmes

La définition interactive des sous-programmes s'apparente tout à fait à celle des programmes. La seule différence concerne la définition éventuelle de paramètres de sortie. Définir *a priori* ces derniers constitue une tâche très abstraite, qui ne peut s'appuyer sur le déroulement de l'exemple. En revanche, définir *a posteriori* les seuls paramètres de sortie est une méthode plus accessible à l'utilisateur final : il suffit d'introduire une seule commande supplémentaire, **objet_accessible**. Lorsque l'utilisateur active cette commande et désigne un objet créé par le sous-programme, celui-ci est automatiquement et implicitement introduit dans la liste des paramètres de sortie du sous-programme, et peut donc ensuite être référencé directement dans le contexte englobant.

À l'usage, il s'est révélé que les besoins majoritaires dans le cadre de notre système CAO se résumaient, d'une part, à manipuler globalement les résultats du sous-programme, et d'autre part, à positionner globalement le résultat de son exécution. Pour faciliter cette deuxième opération, nous avons adjoint à la commande d'appel de sous-programme, qui permet de choisir le sous-programme voulu, puis de définir ses paramètres effectifs à l'aide d'expressions sur les constantes et les variables du programme en cours, la possibilité d'établir la transformation de positionnement.

Notons que l'appel de sous-programme suppose de la part du système de PsE/PbD un changement de mode. En effet, l'appel de sous-programme en phase d'enregistrement engendre le passage en mode exécution, pour permettre l'exécution de l'appel lui-même, c'est-à-dire celle du sous-programme. Le système de PsE/PbD revient au mode enregistrement dès la fin de cette même exécution. Cette alternance des modes enregistrement et exécution est requise également pour les alternatives, et surtout les répétitions.

2.2.3.4.2 Alternatives et expressions

La définition interactive de structures alternatives pose deux types de problèmes, ceux relatifs à la description et à l'évaluation de l'expression de contrôle de la structure, et ceux relatifs à la description du contenu de chaque branche.

Introduire une expression de contrôle exige que l'on dispose d'un ensemble d'opérateurs permettant de définir cette expression. Or, définir un langage de commande textuel incluant ces opérateurs irait à l'encontre des objectifs de simplicité et de transparence pour l'utilisateur de la PsE/PbD. Pour résoudre ce problème, certains systèmes infèrent de plusieurs exemples la nature de l'expression permettant de différencier les cas (Turvy [Maulsby 1993], Tinker [Lieberman 1993]). Cependant, leur pouvoir d'expression demeure extrêmement limité. Dans le domaine de la CAO, un outil interactif puissant répond exactement à ce besoin d'expressions : tous les systèmes CAO proposent en effet une calculette grapho-numérique [Gardan 1991] permettant de définir des expressions complexes, quoique généralement à résultat purement numérique. L'extension de cet outil, pour lui faire calculer également des valeurs booléennes.

Par exemple, « distance (point 1, point 2) > 10 »), constitue une solution simple et naturelle pour l'utilisateur pour la définition des expressions nécessaires aux alternatives.

Pour sa part, la définition interactive proprement dite de l'alternative demande un effort supplémentaire à l'utilisateur. Quatre commandes supplémentaires sont introduites : SI, ALORS, SINON et FIN SI. L'utilisateur ouvre la structure par la commande SI, et définit l'expression par la calculette grapho-numérique. Le système évalue ensuite l'expression et ouvre la branche correspondant à la valeur de l'expression. À l'issue de la définition de la première branche, l'utilisateur peut, soit forcer l'ouverture de l'autre branche (les deux branches sont alors définies sur le même exemple, les résultats apparaissent tous deux sur l'écran, mais ne sont désignables que globalement), soit fermer la structure (FIN SI), modifier les paramètres du programme en cours, et ré-exécuter le programme jusqu'au SI qui, en fonction de la valeur de l'expression, ouvre alors l'autre branche. Deux exemples différents peuvent ainsi être utilisés pour définir les deux branches. Ces deux modalités permettent une grande souplesse d'utilisation, et se sont révélées tout aussi utiles.

2.2.3.4.3 Répétitions

La définition interactive des structures répétitives consiste à définir, outre les actions à effectuer à chaque tour et la condition d'arrêt de la répétition, la relation de récurrence qui unit les objets de deux itérations successives. Si les itérations de collection utilisent une relation de récurrence implicite (« prendre l'objet suivant dans la collection, et lui appliquer les mêmes actions » comme le « Graphical search and replace » ou le « Constraint-based search and replace » de Kurlander [Kurlander 1993b], ou encore Eager [Cypher 1991]), les itérations générales (itérations de récurrence) nécessitent la définition de cette relation. Ceci peut se faire par déduction, comme dans les « Constraints from Multiple snapshots » de Kurlander [Kurlander 1993b], mais les procédés utilisés sont, de l'avis même des auteurs, toujours extrêmement limités [Kurlander 1993a]. L'autre solution, que nous avons choisie, consiste à permettre la définition explicite sur l'exemple de cette relation.

Illustrons ce point à l'aide de l'exemple représenté sur la Figure 1. Il s'agit « d'empiler » des cercles d'un rayon divisé par deux à chaque étage, jusqu'à atteindre un rayon minimum. Le programme peut se résumer à une répétition, pour faire une suite de cercles empilés, puis à une symétrie de l'ensemble. Un pas quelconque de la répétition (sauf le premier) peut se définir ainsi :

construire un premier cercle tangent au cercle du tour précédent et à l'axe central, de rayon égal à la moitié de celui du cercle du tour précédent, puis créer un cercle de centre identique au dernier créé, et de rayon égal à la moitié de son rayon.

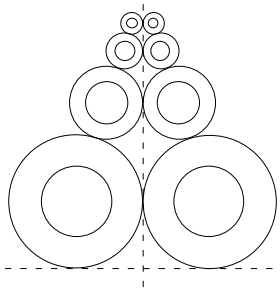

Figure 1 : Exemple de construction complexe

Cette description appelle trois remarques : (1) les objets d'une branche se définissent par rapport aux objets du tour précédent ou du tour courant ; c'est le propre des itérations de récurrence, qui représentent le niveau le plus complexe utilisé en programmation ; (2) la séquence d'actions qui constitue le premier tour est identique à celle des autres tours ; la seule différence est que, en l'absence de tour précédent, les références se font à des objets du contexte englobant : le premier cercle est tangent à deux droites préexistantes ; (3) les relations de récurrence peuvent être entièrement définies en ré-exécutant une deuxième fois les actions du premier tour, et en demandant seulement à l'utilisateur, pour chacun des objets désignés lors du premier tour, si le même objet doit être utilisé pour le tour suivant (c'est le cas de la droite verticale) ou si un objet **construit au premier tour** doit remplacer l'objet du contexte englobant (c'est le cas de la droite horizontale, remplacée par le premier cercle construit lors du premier tour).

Ces trois remarques induisent l'interface utilisateur et les conventions de dialogue que nous avons définies dans le système Like. Deux commandes supplémentaires sont à la disposition de l'utilisateur (RÉPÉTER et JUSQU'À). Après avoir sélectionné la commande RÉPÉTER, l'utilisateur définit le premier tour. Pour ce faire, il accède aux objets du contexte englobant. Il sélectionne ensuite la commande JUSQU'À, sans introduire l'expression de contrôle, qui ne le sera qu'à l'issue du second tour.

Le système passe alors automatiquement en mode exécution, pour définir le deuxième tour. Tout en exécutant automatiquement les commandes, il met en évidence (clignotement) les objets du contexte englobant désignés (ici, la droite horizontale, dans un premier temps). L'utilisateur peut alors seulement (1) désigner la même droite, qui sera alors utilisée dans tous les tours, ou (2) désigner un objet du premier tour (ici le premier cercle créé) ; une relation de récurrence est alors créée, reportée de tour en tour. De même, les expressions utilisées qui mettent en jeu des objets du contexte englobant (ici le paramètre 'rayon') sont mises en évidence ; l'utilisateur peut alors (1) confirmer leur utilisation à l'identique pour tous les tours, ou (2) définir, à l'aide de la calculette grapho-numérique, une nouvelle expression pouvant mettre en jeu à la fois des éléments du contexte englobant, ou des deux tours de la répétition (dans l'exemple, *Rayon_de* (Premier cercle du tour précédent) / 2). Lors de cette exécution du deuxième tour, les références à des objets internes au premier tour sont automatiquement transformées en références au tour courant (par exemple, le centre du premier cercle créé dans le tour courant).

Lorsque tous les objets ont été ainsi re-désignés, le système affiche le message 'JUSQU'À ?' et l'utilisateur introduit l'expression de contrôle, à l'aide de la calculette grapho-numérique, accédant pour ce faire à la fois aux objets des deux tours de la répétition et ceux du contexte englobant. À la fin de cette définition, le système exécute toutes les itérations suivantes, les références utilisées dans les relations de récurrence étant mises à jour de tour en tour.

2.2.4 Vers un environnement de PsE/PbD : de Like à GIPSE

Parmi les préoccupations du Workshop de 1992 [Cypher 1993c], les participants relèvent un certain nombre de points qui concernent l'ensemble des facilités fournies par le système de PsE/PbD pour permettre une bonne utilisation. On peut ainsi citer les différentes façons de créer, visualiser et modifier les programmes, les possibilités d'activation des programmes réalisés, ou encore la généralité des programmes construits. En fait, il s'agit de déterminer des critères permettant d'aller vers de véritables *environnements de programmation* par PsE/PbD.

Tout au long de nos travaux, nous n'avons eu de cesse d'atteindre le même objectif. À partir d'un système dédié à un problème bien particulier (le paramétrage en CAO, 2.2.4.1), nous avons isolé les relations qui doivent être établies entre un système interactif et un système de PsE/PbD susceptible de fournir à l'utilisateur (« End User ») de réelles possibilités de programmation. Par la suite, nous avons exploré les possibilités de génération de programmes neutres (2.2.4.2) ; de là, un véritable environnement de programmation autorisant une mise au point totalement interactive et graphique a pu être défini. Enfin, nous nous sommes plus particulièrement attaché à l'intégration du système de PsE/PbD au sein de son application support (2.2.4.3).

2.2.4.1 Like : une approche au niveau syntaxique

Comme nous l'avons mentionné précédemment, nous avons abordé la PsE/PbD par un angle privilégié, celui du paramétrage en CAO. Celui-ci consiste à permettre la définition de familles de pièces dites « standard » qui peuvent ensuite être assemblées pour constituer les modèles servant de base à la Conception Assistée par Ordinateur.

L'objectif de notre étude consistait à identifier les problèmes soulevés par la PsE/PbD dans le cadre de la CAO, et de proposer des solutions en terme d'exhaustivité de mécanismes de programmation. C'est ainsi que nous avons établi les bases de la solution décrite en 2.2.3.

Le système Like, que nous avons conçu pour mettre en œuvre cette démarche, est ainsi un moniteur de PsE/PbD greffé sur un système de CAO 2D permettant de construire des modèles constitués d'entités simple (points, droites, cercles, ...) éventuellement structurées (notion de pièce). Dans le système CAO support, un soin particulier a été apporté aux diverses opérations de créations par contraintes, et à la définition d'une calculette grapho-numérique capable de calculer, par exemple, *la moitié de la distance séparant le point x du centre du cercle y*.

L'environnement de PsE/PbD fourni dans Like, quoique logiquement « greffé » sur le système CAO lui-même, s'intègre totalement dans le dialogue du système complet, et ne rajoute qu'un nombre minimal de commandes. Des règles strictes, contrôlées par Like, filtrent l'accessibilité aux objets visibles sur l'écran. Ces règles peuvent s'énoncer comme suit :

- Les paramètres du programme doivent être définis par le programmeur et sont désignables ;
- Toute autre valeur numérique ou graphique (x,y correspondant à une position graphique) introduite dans l'exemple est considérée comme constante ;
- Tout objet graphique créé au cours de l'enregistrement du programme reçoit un nom implicite (numérotation) qui permettra au système Like de l'identifier ; cet objet est ensuite désignable ;
- Toute référence (désignation graphique) à un objet créé précédemment est remplacée dans le programme enregistré par l'identifiant de cet objet ;
- En corollaire, toute désignation d'objet non créé au cours du programme est rejetée ;
- L'association des deux ensembles de variables décrits ci-dessus constitue le **contexte dynamique** du programme, formé du **contexte explicite** (paramètres) et du **contexte implicite** (variables à nom implicite).

Ces quelques règles permettent d'obtenir un système d'utilisation très simple, ne possédant que peu de commandes spécifiques de programmation, et dont le comportement est entièrement prévisible (pas d'inférence). Aucune intervention *a posteriori* du programmeur n'est requise pour

modifier le programme généré, dont la représentation en langage spécifique ne lui est normalement pas accessible.

Les limites du système proviennent de sa liaison très stricte avec le contrôleur de dialogue de l'application hôte. Ainsi, tout changement dans le dialogue de cette dernière entraîne l'invalidation des programmes enregistrés. La réduction à son strict minimum de la connaissance de la sémantique de l'application hôte permet en revanche d'envisager l'adaptation du système d'enregistrement sur toute application interactive.

Le système a été validé dans le cadre d'un contrat industriel avec la société Alcatel.

2.2.4.2 EBP : générer en langage neutre

L'étude réalisée par Jean-Claude Potier dans le cadre de sa thèse de doctorat a permis de franchir un nouveau pallier. Toujours dans le domaine du paramétrage en CAO, des relations précises entre le système interactif (système CAO en l'occurrence) et l'environnement de programmation interactive ont tout d'abord été établies, afin de permettre de greffer le second sur le premier.

La deuxième étape de ce travail a consisté à résoudre le problème soulevé par Like, à savoir l'affranchissement du dialogue de l'application hôte. Enfin, sa connaissance de la sémantique de l'application hôte permet au système EBP de générer en langage neutre les programmes construits.

En plus de la réalisation d'un système de PsE/PbD, EBP est un véritable environnement de développement de programmes paramétrés. Il permet en particulier de mettre au point par manipulation directe et visuelle les programmes générés, ce qui pose le problème de la navigation dans le programme ou encore la gestion du UNDO.

Le premier exemple d'application qui a été développé consiste à utiliser les techniques de PsE/PbD pour permettre la création de familles de composants standard portables par des utilisateurs de systèmes CAO. Nous commencerons par établir ci-dessous le cahier des charges du système, puis nous en décrirons les caractéristiques.

2.2.4.2.1 Le cahier des charges d'EBP

La portabilité des catalogues de composants entre systèmes CAO différents est un point d'importance économique majeure pour les utilisateurs de ces systèmes, les fournisseurs de composants, ainsi que les constructeurs de systèmes CAO. Elle devrait augmenter sensiblement le nombre de familles de composants disponibles sur différents systèmes, et par voie de conséquence, augmenter la qualité et la productivité générales. L'approche CAD-LIB a ainsi été developpée [Pierra & Aït Ameur 1994], et constitue la base commune de travaux européens et internationaux de normalisation (CEN/TC310-pr ENV 40004 et ISO/TC184/SC4-ISO 13584 P-LIB).

En 1993, un projet dénommé PLUS (Parts Library Usage and Supply) était lancé. Ce projet, soutenu par l'Union Européenne au sein du programme de Recherche et Développement ESPRIT, avait pour objectifs :

- de développer, sur la base des concepts de CAD-LIB, une spécification complète du format d'échange pour publication en tant que norme ISO 13584 (P-LIB),
- de valider cette spécification à travers le développement d'un ensemble complet d'outils pré-industriels, capables de générer ou d'utiliser ce format d'échange.

Au-delà d'un modèle de données orienté-objet pour l'échange des données des bibliothèques de composants, le projet devait développer une approche permettant l'échange de familles de modèles géométriques. Au début du projet, les technologies variationnelle et paramétrique n'apparaissaient pas suffisamment mûres pour permettre le développement d'une norme de format d'échange pour les modèles paramétriques. Aussi, l'approche choisie a-t-elle été résolument « conservatrice ». Elle a consisté à développer une Interface de Programmation d'Application (API) standard (maintenant disponible sous le nom de ISO 13584-31) associé à un couplage FORTRAN.

Tous les systèmes CAO supportant une implémentation de cette API seraient ainsi capables d'exécuter un programme FORTRAN se référant à cette API.

Cette approche était cependant moins conservatrice qu'elle n'en avait l'air au premier coup d'œil : le projet incluait également le développement d'un système PsE/PbD destiné à générer ces programmes au moyen d'interactions purement graphiques.

Ce contexte définit le cahier des charges de la conception du système EBP. (1) Le processus de génération doit être déterministe, et contrôlé pleinement par l'utilisateur : une seule forme doit être générée pour un ensemble quelconque de valeurs pour les paramètres d'entrée, et cette forme doit effectivement correspondre à un composant. (2) Toutes les familles descriptibles à l'aide de programmes conventionnels doivent pouvoir être construites avec le système. (3) Le système doit pouvoir générer une représentation externe de sa structure de données interne dans le format défini par l'API normalisée (i.e. un programme FORTRAN faisant appel à l'API normalisée).

On peut noter que, parmi les systèmes paramétriques actuels (le premier point ci-dessus exclut le recours à toute forme d'inférence ou d'heuristique, donc exclut l'approche variationnelle), aucun d'entre eux ne remplit en totalité les conditions émises en (2) et en (3).

Le système EBP (Example Based Programming) [Potier 1995] est un système de CAO 2D. Il manipule des entités géométriques simples (points, droites infinies, segments, cercles, arcs, etc.) et des entités structurées (courbes, surfaces planes et ensembles structurés). La plupart des contraintes résultant des règles du dessin technique sont supportées. EBP fournit de plus une calculatrice puissante qui permet de combiner nombres et entrées graphiques dans des expressions grapho-numériques. Enfin, EBP permet la construction du modèle à travers une interface basée sur la norme X-MOTIF, et tourne sur stations Sun-Solaris[2] et DEC-Alpha[3]. Il a également été porté sur PC-WINDOWS NT, avec la couche graphique Tcl/Tk.

2.2.4.2.2 Un système totalement déterministe

L'ambiguïté des constructions géométriques n'est pas spécifique des systèmes variationnels [Martin 1995]. Elle est en fait intrinsèque aux problèmes géométriques non linéaires, par exemple lorsqu'ils font intervenir des contraintes sur des cercles.

Ainsi, construire une ligne à partir d'un point et devant être tangente à un cercle conduit à deux solutions possibles, comme le montre la Figure 2 :

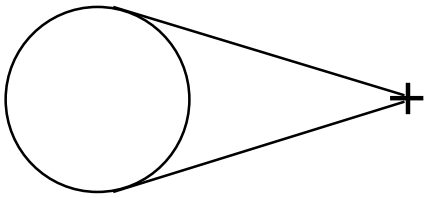

Figure 2 : Les deux solutions possibles pour une ligne tangente à un cercle

Ce problème est résolu à la perfection dans un système interactif. La plupart des systèmes CAO utilisent la position de la souris au moment de la désignation de chaque objet pour distinguer les solutions possibles. Ils supposent alors que l'utilisateur connaît approximativement la solution qu'il désire. Par exemple, dans le cas de la Figure 2, le clic souris permet de choisir la solution supérieure (Figure 3).

2 Sun Inc., USA
3 Digital, USA

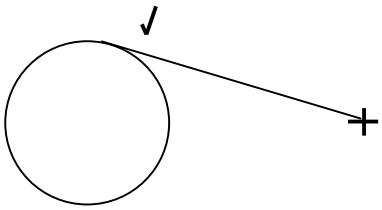

Figure 3 : Résolution par pointé graphique

Si cette convention de dialogue peut être utilisée au niveau interactif, et donc, en particulier, lors de la conception de l'exemple dans un système paramétrique, elle ne peut être conservée en l'état dans le programme où, pour des valeurs différentes des paramètres, la position peut correspondre à des solutions différentes. Ce problème n'est pas spécifique de la PsE/PbD, mais se retrouve dans de nombreux cas de programmation géométrique [Roller 1990].

Dans ces derniers cas, des éléments de discrimination de solutions hors-contextes ont pu être définis. Ainsi, dans l'API cible, la résolution des ambiguïtés est-elle assurée à l'aide d'informations topologiques supplémentaires : l'orientation des entités géométriques et la notion d'intérieur/extérieur pour les cercles. Par exemple, la Figure 4 montre la solution unique de la fonction de création d'un segment passant par un point et un cercle de la Figure 3 :

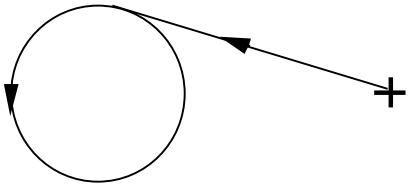

Figure 4 : Résolution des ambiguïtés par orientation

EBP assure de cette façon la transformation des informations contextuelles capturées lors de l'utilisation interactive (la position de la souris lors de la désignation) en informations hors-contexte enregistrables dans un programme général.

Dans le système EBP, toutes les entités sont orientées par leur construction. Les segments sont orientés depuis leur origine jusqu'à leur extrémité, les cercles sont orientés en sens inverse des aiguilles d'une montre, etc. Durant l'enregistrement du programme, le système transforme l'information contextuelle « par proximité » en mécanisme d'orientation de la façon suivante :

- Le système calcule la bonne construction par proximité,
- Ensuite, le système calcule la bonne orientation.
- Si cette orientation est consistante avec la solution (comme dans la Figure 4), le système enregistre l'action sans modification.
- Sinon, le système enregistre la séquence suivante : changer l'orientation du cercle, dessiner le segment, puis changer à nouveau l'orientation.

Ce mécanisme, qui reste caché à l'utilisateur final, est totalement déterministe.

2.2.4.2.3 EBP, système de Programmation sur Exemple

La programmation visuelle sur exemple est réalisée dans EBP via un mode d'enregistrement des appels de fonctions systèmes. Ceci signifie que, contrairement à certains systèmes paramétriques où les programmes sont directement basés sur leur valeur d'exemple (i.e. la fonction **line_2_points** fait directement référence au point exemple), les programmes au sein d'EBP (où ils sont nommés « instances ») sont séparés des exemples. Les relations entre valeurs d'exemple et variables du programme sont fournies au travers **du contexte dynamique** du programme. Ce mécanisme renforce l'indépendance entre le gestionnaire PsE/PbD, qui manipule les variables, et le système de CAO où les valeurs d'exemple sont des références à la base de données CAO. Lorsque le programme est ré-exécuté (i.e. lors de modifications), les variables EBP ne sont pas modifiées, mais leurs contenus (adresses en base de donnée stockées au sein du contexte) sont mis à jour.

Après l'activation du **mode enregistrement**, le système EBP « espionne » l'utilisateur et construit une instance. Le passage en **mode utilisation** permet à l'utilisateur d'exploiter cette instance.

Les seules commandes additionnelles vis-à-vis d'un système de CAO traditionnel sont destinées à charger, nommer, sauver et exécuter des instances (commandes SAVE, NAME, LOAD, APPLY), ainsi qu'à définir, lire, écrire et saisir les paramètres (commandes DEFINE, READ, WRITE, ENTER). L'adjonction de structures de contrôle nécessite d'autres commandes, décrites dans la section suivante.

Une session typique d'EBP serait la suivante : après l'analyse de la pièce (quels sont les paramètres ? Où sont les dépendances ?... tâches courantes, que les dessinateurs sont habitués à réaliser à chaque fois qu'ils envisagent de réaliser un modèle CAO), l'utilisateur commence l'enregistrement PsE/PbD. Il définit les paramètres, puis dessine un exemple en utilisant les paramètres au lieu de valeurs directes. La commande DEFINE ouvre une fenêtre, où l'on fournit un nom de paramètre (par exemple, « longueur »), lequel est affiché à chaque exécution du programme. La commande ENTER permet l'introduction de la valeur des paramètres pour l'exemple. Ces valeurs sont saisies via l'interface du système de CAO, et leur type définit le type des paramètres. Les commandes READ et WRITE permettent la lecture (resp. l'enregistrement) de valeurs à partir d'un fichier, lequel sera associé à l'instance. Cette fonction est utilisée pour enregistrer les valeurs autorisées de paramètres pour des familles de pièces. Dès que les paramètres sont définis, ils sont affichés au sein d'un menu où l'utilisateur peut les désigner, par exemple pour définir une expression à l'aide de la calculatrice grapho-numérique (ex. : « longueur x 2 »).

Lorsque l'exemple est complet, l'utilisateur peut sauver l'instance résultante, changer quelques valeurs de paramètres et commander une nouvelle exécution. Enregistrer sur fichier des valeurs de paramètres est aisé, permettant des procédures de test rapides. Les instances enregistrées sont incluses dans un menu, et sont utilisables avec un minimum d'effort.

La commande LOAD sélectionne une instance et la commande APPLY l'exécute. Ces commandes peuvent être sélectionnées à la fois en mode utilisation et en mode enregistrement. Dans le premier cas, un modèle sera créé au sein de la base de données du système CAO. EBP correspond alors à un système de macro-par-exemple. Dans le second cas, la commande est enregistrée comme un appel de sous-programme dans l'instance en cours, et EBP assure le passage des paramètres. Dans les deux cas, après la sélection de la commande, EBP affiche chaque paramètre et attend une valeur. Cette valeur est définie à l'aide de l'interface du système CAO. Ceci signifie que, lors de l'exécution de l'instance en mode enregistrement, la définition de la valeur du paramètre effectif est constituée d'une expression quelconque mettant en jeu des entités ou des paramètres du programme appelant. Il s'agit d'un réel « passage de paramètres » comme il existe dans les langages classiques.

2.2.4.2.4 Toutes les structures de contrôle

Le système EBP comporte des structures de contrôle complètes. Plus précisément, les sous-programmes, déjà évoqués ci-dessus, les alternatives et les répétitions sont totalement disponibles. La consistance du contexte des programmes est gérée entièrement par le système, et permet une utilisation correcte de ces structures.

Les alternatives et les répétitions supposent la définition d'expressions booléennes, ce qui est fait par l'intermédiaire des calculatrices grapho-numérique et logique. Les deux branches des alternatives peuvent être définies soit dans un mode consistant avec les données (en exécutant l'instance avec de nouveaux paramètres), soit dans un mode inconsistant (en dessinant les deux solutions avec le même ensemble de paramètres).

Plusieurs schémas d'itération sont fournis. L'itération d'ensembles (« set iteration ») est disponible, pour des objets sélectionnés à l'aide d'un rectangle élastique, ou pour les transformations géométriques multiples, lesquelles sont courantes dans les systèmes CAO. Comme dans le cas de la

résolution des ambiguïtés, les informations dépendantes du contexte (les deux coins du rectangle élastique par exemple) sont traduites en informations indépendantes du contexte (l'ensemble des entités incluses dans le rectangle élastique). De façon plus générale, des structures telles que *Répéter_n_fois...*, *Tant que...*, ou *Répéter... Jusqu'à ...* sont aussi disponibles. Elles permettent la définition d'itérations de récurrence, et ce d'une manière totalement interactive.

2.2.4.2.5 EBP : un véritable environnement de programmation

Le système EBP est un environnement de programmation complet, qui fournit toutes les facilités de mise au point, dans un mode totalement basé sur exemple. Chaque interaction avec le programme se fait à travers l'exemple d'exécution. Les programmes générés sont montrés dans une fenêtre spécifique affichée uniquement sur demande de l'utilisateur. Un menu spécial permet la ré-exécution des programmes.

- **Modification des programmes et UNDO/REDO intelligent**

Toutes les modifications de programme sont possibles, tant en mode mise au point qu'en création de programme. Des UNDOs successifs sont possibles, de manière illimitée, et l'ajout ou la suppression d'actions est toujours possible, et ce de manière consistante avec le reste du programme. Ainsi, quand la ré-exécution d'une partie de programme résultant d'une modification fait référence à des entités non-existantes du fait des modifications, le système EBP demande à l'utilisateur de remplacer les désignations d'objets.

- **Mise au point visuelle**

La représentation textuelle des programmes n'étant jamais supposée visible, la mise au point de programmes « virtuels » pourrait apparaître difficile. En fait, cette représentation n'est pas nécessaire, l'exemple constituant toujours une interface d'entrée/sortie pour le dialogue avec le programme.

Comme tout environnement de mise au point (« debugger »), le système EBP permet d'exécuter un programme pas à pas, ou jusqu'à sa fin. Il permet également de ré-exécuter le programme **jusqu'à ce qu'une entité soit créée**. L'utilisateur désigne graphiquement l'entité, et le programme s'exécute jusqu'à ce pas. Les modifications et l'exécution pas à pas sont alors possibles.

- **Génération de programmes portables**

Le système EBP a été conçu pour permettre la production de bibliothèques de composants standard, selon la norme ISO 13584-31. Ces programmes, qui peuvent demeurer cachés à l'utilisateur, ne peuvent être modifiés textuellement sous l'environnement EBP. Des travaux sur la génération en langage Java ont été récemment conduits.

EBP a fait l'objet d'un contrat de valorisation ANVAR qui a permis de l'industrialiser complètement à l'issue du projet PLUS. Deux copies d'écran d'EBP sont disponibles en annexe.

2.2.4.3 GIPSE : extension et spécialisation d'un système

L'une des limites du système EBP concerne l'intégration des programmes générés par PsE/PbD au sein de l'application. Les objectifs d'EBP étant centrés sur la génération de programmes neutres, ces derniers sont traités comme des entités externes qui peuvent être activées par des mécanismes de dialogue spécifiques.

Une extension de ce travail a été effectuée par Guillaume Patry dans le cadre de sa thèse soutenue en Mars 1999 [Patry 1999a].

Comme nous l'avons démontré avec les deux autres systèmes, créer de nouvelles fonctions par programmation par démonstration est un processus désormais maîtrisé. Cependant, utiliser ces nouvelles fonctions au sein du système relève généralement d'un dialogue spécialisé, qui consiste à choisir une macro parmi une liste de macros disponibles. Dans le meilleur des cas, un raccourci clavier permet d'activer directement une macro. L'exécution de la macro engage alors son propre dialogue pour acquérir ses paramètres.

Obtenir une intégration complète des macros construites par l'utilisateur suppose que le système puisse les activer de la même façon que les fonctions de base (fonctions « codées en dur »). D'un point de vue technique, il s'agit pour le contrôleur de dialogue de pouvoir appeler de la même façon des fonctions qui sont syntaxiquement très différentes. Du point de vue du dialogue lui-même, il s'agit de pouvoir intégrer complètement et harmonieusement l'appel de ces nouvelles fonctions.

Des solutions ont été proposées, qui ont conduit au système GIPSE, qui sera par ailleurs beaucoup plus détaillé dans la section 3.2.4. Elles ont ainsi permis une totale intégration des nouvelles actions dans le dialogue de l'application. Ces actions peuvent utiliser le résultat des autres actions et ont un comportement totalement identique aux actions natives de l'application initiale.

De par sa structure, GIPSE forme ainsi la base d'un environnement de conception. De la même manière que Microsoft® ACCESS™ permet à un utilisateur de créer une nouvelle application en se basant sur une base de donnée relationnelle, GIPSE autorise un développeur à concevoir une application spécialisée en se basant sur une application de conception technique générique.

Là où un utilisateur d'ACCESS™ construit son application en définissant requêtes et masques de saisie, c'est-à-dire appels au noyau fonctionnel et présentation, GIPSE permet la construction d'une nouvelle application en définissant les nouvelles fonctions à partir de celles déjà existantes, puis en les plaçant dans l'interface de dialogue. L'utilisateur peut modifier le dialogue en supprimant ou réordonnant ces fonctions. À toute modification correspond une re-génération dynamique du dialogue du système.

2.2.5 En résumé

Dans cette section, nous appliquons dans un premier temps la grille d'évaluation de Cypher [Cypher, Kosbie, & Maulsby 1993] aux trois systèmes auxquels nous avons contribué. Puis, dans un deuxième temps, nous relevons l'ensemble des publications que nous avons réalisées ou induites sur les travaux présentés.

2.2.5.1 Grille d'évaluation

Dans la taxinomie que nous avons présentée, et à l'instar des systèmes paramétriques, les systèmes Like et GIPSE se situent dans la catégorie des environnements de programmation interactive impérative graphique sur exemple. EBP, quant à lui, se situe dans la même catégorie pour ses capacités de construction de programme, mais se situe également dans la catégorie des systèmes compilés, car il permet de générer des programmes, dont la compilation à l'aide d'un autre système CAO permettra l'exécution.

Avec la grille d'évaluation de Cypher, nous pouvons bien davantage affiner les caractéristiques de ces systèmes.

- **Domaines d'utilisation et utilisateurs**

Le **domaine d'utilisation** de Like et EBP est très précisément le paramétrage en conception mécanique 2D. Le domaine de GIPSE est celui des outils de conception et de réalisation d'interfaces utilisateurs, appliqués à la conception assistée par ordinateur.

Dans tous les cas, les **utilisateurs** sont des utilisateurs finaux du système, mais d'un genre un peu particulier : étant concernés par des tâches d'édition et de conception, ils doivent être considérés comme des experts, tant dans le domaine sous-jacent (CAO ou conception d'interface) que dans la maîtrise de leur outil. Les systèmes Like et surtout EBP, dans une moindre mesure GIPSE, supposent une bonne pratique des applications à dialogues structurés (voir 3.2.1).

- **Interactions de l'utilisateur**

Tous les systèmes utilisent le même principe d'un enregistrement volontaire des programmes : l'utilisateur décide quand il veut utiliser la PsE/PbD. En l'absence d'inférence, les systèmes

attendent de l'utilisateur qu'il précise son intention. Dans certains cas (répétitions par exemple), ils demandent à l'utilisateur des précisions.

La description des données est réduite à la simple différenciation des constantes, variables et paramètres. Les structures de contrôle sont complètes dans Like et surtout EBP, mais n'ont pas encore été introduites dans GIPSE. Like et EBP fournissent une représentation textuelle non modifiable des programmes construits. Dans Like, il ne s'agit que d'un contrôle pour la mise au point de Like lui-même. Cette représentation est très difficilement compréhensible par l'utilisateur final. Dans EBP, la représentation est sous forme d'un langage de programmation (FORTRAN ou Java), mais n'est pas destinée à être modifiée par l'utilisateur. Enfin, GIPSE ne fournit pas de représentation externe du programme généré.

Dans Like et EBP, l'invocation des programmes se fait au moyen de commandes particulières au système de PsE/PbD. Dans GIPSE, l'invocation est totalement identique à celle des fonctions originelles du système. Leur utilisation au sein du dialogue est totalement transparente et cohérente avec le reste du système.

Sur le plan du contrôle d'exécution et des possibilités de mise au point, seul EBP présente un environnement riche. Il permet une exécution pas à pas, dans les deux sens, avec points d'arrêt. Il est également le seul système à notre connaissance à permettre un contrôle de l'exécution sur exemple : on peut demander au programme de s'exécuter jusqu'à un certain point, déterminé sur l'exemple. Enfin, il permet une modification « intelligente » du programme en cours de mise au point ; EBP gère les incohérences amenées par les modifications de l'utilisateur.

- **Inférence**

Comme nous l'avons expliqué, nous avons délibérément choisi d'éviter toute inférence. Les procédés utilisés sont tous algorithmiques et déterministes. Tout au plus peut-on voir dans certains des choix d'interaction des « inférences » du système, que nous avons rangés dans la catégorie des conventions de dialogue. Ainsi, EBP est-il capable de gérer une répétition implicite à travers une construction bien connue en conception mécanique, la transformation géométrique répétitive.

- **Connaissances du domaine**

La connaissance du domaine est assez différente selon les systèmes : Like n'utilise aucune information en provenance du noyau fonctionnel de l'application. Il connaît néammoins la signification de types de base, comme les numériques et les booléens, qui lui permettent de contrôler les structures alternatives et répétitives. Il utilise abondamment les capacités d'interprétation du système hôte, comme par exemple pour évaluer une expression grapho-numérique.

EBP a une connaissance beaucoup plus grande de la sémantique du domaine. En effet, il a pour but de produire des programmes neutres, et doit donc de ce fait connaître la sémantique des actions invoquées. Cette caractéristique se traduit par une beaucoup plus grande neutralité des programmes construits. Alors qu'une simple modification dans le dialogue de Like rendait inopérants les programmes générés précédemment, les programmes d'EBP sont destinés (et ont été utilisés) à être exploités par des systèmes totalement différents.

GIPSE connaît simplement la sémantique du contrôleur de dialogue de l'application. Il ne s'intéresse pas au noyau fonctionnel en tant que tel. Ses programmes sont « propriétaires », et ne peuvent exécutés que sur le système qui a servi à les construire.

2.2.5.2 Publications référencées

En conclusion, on peut dire que les travaux que nous avons menés sur la PsE/PbD ont permis d'atteindre deux grands objectifs :

- Nous avons démontré à travers un domaine particulier, la CAO, que la PsE/PbD pouvait se révéler très utile dans des applications de dimension réelle, et à usage professionnel. EBP a ainsi été industrialisé et distribué.
- Nous avons montré comment réaliser des applications de PsE/PbD totalement déterministes utilisant toutes les structures de contrôle de la programmation impérative.

Nous avons montré également comment un tel système pouvait se greffer sur un système interactif quelconque, apportant à ce dernier de façon générique les possibilités d'automatisation des tâches répétitives, qui font globalement défaut à ces systèmes.

Les travaux présentés dans cette section ont donné lieu à plus de quinze publications :

Like et les principes de base des structures de contrôles en PsE/PbD ont été décrits dans [Girard 1992 ; Girard & Pierra 1990 ; Girard & Pierra 1993].

L'extension du concept de PsE/PbD a fait l'objet de plusieurs publications [Girard & Pierra 1995a ; Girard & Pierra 1995b ; Potier & Girard 1993 ; Potier, Girard, & Pierra 1993]

L'application des techniques de PsE/PbD au domaine de la CAO a été plus particulièrement décrite dans [Girard, Pierra, & Potier 1997 ; Pierra, Potier, & Girard 1994 ; Potier, et al. 1995 ; Potier, et al. 1997] alors que le système EBP a été décrit dans [Pierra, Potier, & Girard 1996 ; Potier et al. 1995] et a fait l'objet d'une cassette vidéo présentée dans [Girard, Potier, & Pierra 1996].

Enfin, les aspects PsE/PbD de GIPSE ont été présentés dans [Girard, et al. 1997] et dans [Patry 1999a ; Patry & Girard 1997a]

2.3 Perspectives

Nos travaux dans le domaine de la PsE/PbD ont participé à l'évolution des systèmes CAO « paramétriques » qui se sont généralisés au cours des années 1990 et qui ont réalisé des progrès considérables. Les perspectives qu'ils ont ouvertes relèvent essentiellement de la diffusion des résultats obtenus dans de nouveaux domaines d'application. En dehors de la CAO, la PsE/PbD souffre en effet d'un manque de reconnaissance dû principalement au faible nombre d'applications vraiment industrielles des techniques présentées.

2.3.1 Application de la PsE/PbD à la simulation de croissance

De nouveaux champs d'application se présentent dans le cas où les objets graphiques sont fortement structurés comme les fractals ou les objets arborescents. Lorsque l'on veut décrire des comportements d'objets, par exemple pour la simulation de croissance, on se base généralement sur une modélisation sous-jacente qu'il convient de programmer. Alors que le résultat souhaité est éminemment visuel, les techniques de description se résument à des méthodes très textuelles. Il s'agit de langages spécialisés (grammaires) ou de programmes classiques utilisant une modélisation sous-jacente spécifique. Le processus d'élaboration de ces programmes est typiquement basé sur un cycle « compilation / validation / test ». À partir du résultat escompté, le plus souvent représenté visuellement (sous forme de photos par exemple), le concepteur doit imaginer le comportement (« la loi de croissance ») permettant de l'atteindre, l'abstraire, le coder, le compiler, puis le tester, pour s'apercevoir en fin de compte que le résultat obtenu n'est pas celui attendu.

Quoi de plus naturel que d'imaginer interagir directement avec l'exemple construit pour définir graphiquement la croissance maximale, le point d'inflexion d'un comportement, ou encore un événement fortuit interrompant la loi normale (rupture d'une branche dans le cas de la croissance d'un arbre par exemple). Utiliser la PsE/PbD pour concrétiser au plus tôt le résultat attendu semble donc une idée séduisante. L'enjeu scientifique consiste à abstraire un comportement de structure arborescente dynamique à partir d'une spécification graphique.

Alors que les opérations fondamentales de la croissance sont des répétitions, il convient de trouver des mécanismes qui permettent à l'utilisateur de disposer de plusieurs interprétations possibles de ses interactions pour engendrer des comportements différents. Les mécanismes d'inférence sont toujours limités à un nombre fini de cas, alors que l'évolution en simulation de croissance peut faire l'objet d'un nombre quelconque de cas. Afin de dépasser cette limite, doit être fournie à l'utilisateur la possibilité de créer ses propres interprétations des interactions, et de les traduire en comportements du système.

Ce travail a été engagé dans le cadre d'une thèse de Doctorat en liaison avec P. Liehnardt et G. Pierra.

2.3.2 Apprentissage de la programmation

Alors que le thème de l'apprentissage de la programmation est à la base des premiers travaux en PsE/PbD ([Bauer 1979 ; Lieberman 1993]), il est apparemment sorti des préoccupations des différents auteurs travaillant aujourd'hui dans le domaine.

Pourtant, là encore, il paraît naturel d'imaginer que travailler directement avec l'exemple permettra de diminuer l'effort d'abstraction nécessaire à la tâche de programmation. Comme nous l'avons remarqué lors de notre étude de la PsE/PbD en CAO, l'avantage de l'exemple n'est réellement déterminant que lorsque les objets manipulés sont intrinsèquement graphiques. Lorsque l'on manipule des objets non graphiques, leur visualisation sous une forme permettant l'interaction induit généralement une distance supplémentaire qui annule les avantages obtenus par ailleurs. Si ceci est pertinent pour la manipulation de programmes séquentiels classiques, deux points peuvent être avancés.

Aux premiers stades de la programmation, abstraire un comportement sous forme de séquences, d'alternatives et de répétitions est souvent difficile. Adapter un système comme Eager [Cypher 1993a], qui détecte les répétitions de l'utilisateur, à un but d'apprentissage semble tout à fait envisageable. En allant plus loin, on peut penser qu'un tel système serait à même d'aider à l'apprentissage de l'algorithmique. Qu'il s'agisse de tris ou de recherches de motifs, l'interaction directe avec l'exemple s'avère pertinente. La traduction de ces interactions en algorithme et leur visualisation dynamique par l'utilisateur (l'étudiant en apprentissage) doit permettre à celui-ci de comprendre ce qu'il réalise réellement.

L'apprentissage de la programmation événementielle est un autre sujet pour lequel la PsE/PbD peut apporter des solutions intéressantes. Là où ce mode de programmation, basé sur la réaction aux événements de l'utilisateur, oblige à éclater l'algorithmique dans un nombre important d'unités de programmation, l'interaction directe avec l'exemple permet de concrétiser le comportement. Certaines idées de Gamut [McDaniel 1999] sont ainsi susceptibles d'être étendues, mais dans un but complètement différent de celui de son auteur. Il s'agit alors à l'utilisateur de comprendre le programme construit, et d'en faciliter l'extension.

2.3.3 Application à l'ingénierie des systèmes interactifs

De nombreux travaux ont cherché à intégrer les techniques de PsE/PbD à l'ingénierie des interfaces homme-machine. Le plus souvent, un usage abondant de l'inférence a été fait, ce qui rend leur comportement non déterministe et restreint donc leur usage à des exemples d'école. Dans la suite de nos travaux sur GIPSE, il nous semble intéressant de pousser plus avant la réflexion menée actuellement au laboratoire (thèse de Guillaume Texier) sur l'intégration dans un système interactif de nouvelles classes d'objets construits sur exemple. Les aspects pertinents pour le dialogue de la modélisation des objets du domaine doivent être déterminés. C'est ce à quoi aspirent des systèmes comme TRIDENT [Bodart, et al. 1995a] ou Mobi-D [Puerta 1997], dans des domaines où la modélisation est parfaitement définie. HandsOn [Castells & Szekely 1999] essaie ainsi de définir par des contraintes externes transmises par PsE/PbD les relations propres aux objets du noyau fonctionnel pour générer leur présentation.

De la même façon, les techniques de PsE/PbD ont été utilisées pour traduire sous forme de contraintes des schémas effectués en tracé libre. [Landay & Myers 1995], mais également [Lecolinet 1999] en sont des exemples. Ils se limitent cependant à la description externe de l'interface, comme l'identification des widgets ou leur positionnement. Les objets dynamiques, et tout particulièrement ceux issus du noyau fonctionnel de l'application, en sont exclus. Le rapprochement de ces deux séries de travaux, et leur extension par utilisation de la PsE/PbD, sont à même d'apporter des solutions nouvelles.

2.3.4 Application à l'édition textuelle

La PsE/PbD a été utilisée de façon expérimentale dans plusieurs systèmes pour des manipulations textuelles. Tels [Witten & Mo 1993] et Tourmaline [Myers 1993b] en sont des exemples. Dans le cadre de l'édition de documents, et plus particulièrement des travaux sur la publication de catalogues de composants industriels (travaux menés au LISI), il semble intéressant d'appliquer la PsE/PbD aux tâches répétitives qui consistent à baliser un texte pour le structurer à l'aide de SGML. Une approche comme celle de Grammex [Lieberman, Nardi, & Wright 1998] semble tout à fait intéressante. Mais son principe de base d'utilisation de l'inférence en limite l'usage. La définition de mécanismes déterministes purement algorithmiques semble pouvoir s'adapter à cette problématique.

Chapitre 3
Outils d'ingénierie du développement des systèmes interactifs

Notre deuxième domaine de recherche concerne l'ingénierie des systèmes interactifs, et plus particulièrement la gestion du dialogue de ces applications. Nous ne détaillerons pas ici les travaux menés en commun avec Laurent Guittet autour des architectures des systèmes interactifs qui ont conduit à la définition du modèle H^4 et à son implémentation sous la forme des interacteurs hiérarchisés (nous présenterons seulement certains aspects du modèle des interacteurs de dialogue au chapitre 4), et ne reviendrons pas sur les définitions de modèles d'architecture et de langages de contrôle de dialogue (voir à cet égard [Patry 1999a]). En revanche, nous présentons ci-dessous le domaine des outils de conception d'interfaces (3.1), puis nous montrons certains aspects de notre contribution à ce domaine (3.2)

3.1 Le domaine de recherche

Les utilisateurs de systèmes interactifs exigent de plus en plus de pertinence des interfaces de leurs logiciels. Ces derniers doivent être plus faciles à utiliser, mais doivent disposer d'une puissance toujours plus grande. La conception et le développement des interfaces constituent ainsi une tâche fastidieuse et coûteuse [Johnson & Johnson 1993 ; Myers 1993c].

Il est aujourd'hui exclu de construire une interface utilisateur *ex nihilo*. Pour des raisons techniques (homogénéité de l'interface, complexité de programmation), et économiques (temps de réalisation), de nombreux outils ont été élaborés. Cette section en présente les grandes catégories. Même si nous tirons principalement nos définitions des travaux de Myers [Myers 1995] réactualisés en 1996, nous avons choisi de faire une présentation selon la logique de [Coutaz 1990], basée sur les niveaux de services offerts pour le codage des interfaces, en détaillant tout d'abord les *boîtes à outils* ou « *toolkits* » (3.1.1) puis en présentant les *squelettes d'applications* ou « *application frameworks* » (3.1.2). Ensuite, nous présentons les outils plus évolués, SGIU/UIMS (3.1.3) d'une part, et Model-Based Systems (3.1.4) d'autre part, avant de clore par un court commentaire sur l'état actuel du domaine.

3.1.1 Boîtes à outils, ou « toolkits »

Les systèmes de fenêtrage peuvent être aujourd'hui considérés comme le niveau de base standard sur lequel sont construites les interfaces graphiques modernes. Cela étant, ils se contentent de fournir une interface de programmation (API pour « *Application Programming Interface* ») de très bas niveau. Les *boîtes à outils* ou « *toolkits* » constituent la première couche d'outils située au-dessus de cette API. Il faut d'ailleurs remarquer que beaucoup de boîtes à outils ne sont pas franchement isolées de la couche de base du système de fenêtrage [Fekete 1996b].

Le principe de base des boîtes à outils est double. Il consiste (1) à définir des objets de gestion de l'interaction, communément appelés *widgets*, et (2) à homogénéiser à la fois l'aspect et le comportement (« *Look & Feel* ») des interfaces construites au travers des objets manipulés, mais également des primitives de base du système de fenêtrage accessibles à travers l'API. Les niveaux d'abstraction peuvent aller de la représentation d'objets élémentaires à la gestion de certains éléments du dialogue. Cependant, la gestion de ce dernier est laissée à l'application.

Une boîte à outils présente certains avantages. En premier lieu, elle donne un style à l'interface, qui intègre des critères ergonomiques. Ensuite, elle permet d'obtenir une homogénéité visuelle entre applications. Enfin et surtout, elle est en général très puissante. Une boîte à outils facilite la programmation des différents éléments visualisés par une interface graphique.

En contrepartie, l'utilisation d'une boîte à outils présente de nombreux inconvénients, le premier d'entre eux étant une indubitable difficulté d'utilisation. Il est nécessaire d'être informaticien pour s'en servir, ce qui exclut de nombreuses personnes pouvant êtres amenées à travailler sur l'interface (spécialistes des interfaces homme-machine, utilisateurs finaux). Le temps d'apprentissage est en outre important. La maintenance de l'application est rendue difficile, les appels à la boîte à outils étant généralement dispersés dans tout le code de l'application, puisque celle-ci gère le dialogue. Enfin, la portabilité entre boîtes à outils différentes est problématique [Meinadier 1991].

La plupart des boîtes à outils évoluées fournissent des services sous forme d'un ensemble d'objets de présentation standard et ayant un comportement prédéfini, tels que bouton, menu, surface de visualisation (« canvas »), etc. On retrouve dans cette catégorie des boîtes à outils telles que Motif, Win32, ou Tk. Ces boîtes à outils sont générales, et peuvent être employées pour le développement d'applications de tout genre. Cependant, l'emploi d'un style d'interaction autre que celui défini par les objets standard doit être codé. Par exemple la manipulation directe n'est pas directement intégrée dans ces boîtes à outils. Ainsi, X_{TV} [Beaudouin-Lafon, Berteaud, & Chatty 1990 ; Chatty 1992] et Amulet sont-elles des boîtes à outils adaptées à la manipulation directe. Amulet dispose en outre de facilités pour le traitement de la reconnaissance de gestes. Whizz [Chatty 1992], boîte à outil construite au-dessus de X_{TV}, est, elle, destinée à l'animation. À l'inverse, il existe un certain nombre de boîtes à outils basées sur des paradigmes différents, qui favorisent d'autres styles d'interaction.

Plus récemment, XXL [Lecolinet 1996], une boîte à outils construite au-dessus de Motif, permet comme Amulet d'aller plus loin dans la construction d'applications, et ainsi d'allier des outils graphiques de construction d'interface (cf. 3.1.3.1) à une bibliothèque de widgets, décrite dans un langage de description compatible avec le C ANSI. De ce fait, elle allège singulièrement la tâche de programmation.

3.1.2 Squelettes d'application

Les boîtes à outils fournissent au développeur des objets de présentation, mais lui laissent le soin de gérer les séquences de dialogue [Meinadier 1991]. Il est amené à programmer certaines séquences de code ou structures un grand nombre de fois, dans des applications différentes. Ainsi, s'est-on rendu compte [Myers 1995] à l'usage de la « Macintosh toolbox » que les programmeurs éprouvaient de grandes difficultés pour nommer les différentes fonctions fournies dans la boîte à outils, tout en respectant les règles de conception permettant d'assurer un aspect et un comportement uniformes des interfaces (« Apple guidelines »).

Il est donc apparu intéressant de fournir au programmeur une structure d'application interactive adaptable, véritable guide de réalisation de l'interface. Il ne lui reste plus qu'à « remplir les blancs » dans les zones de communication entre application et interface, en supprimant les portions inutiles, modifiant celles inadaptées et ajoutant de nouvelles fonctions.

L'application est alors vue comme une série de primitives appelées par l'interface. On aboutit à une structure de contrôle externe.

L'utilisation d'un squelette d'application facilite le développement d'une plus grande partie de l'ensemble d'une application interactive que les simples boîtes à outils, puisque la gestion d'une partie du dialogue y est intégrée. Ceci permet au développeur de se concentrer sur les fonctions de l'application. Cette méthode garantit également une homogénéité élevée au niveau de l'interface utilisateur aussi bien qu'entre applications. Tout compte fait, on estime qu'un squelette d'application, lorsqu'il est bien adapté au domaine visé, diminue le temps de développement d'un facteur quatre à cinq par rapport aux seules boîtes à outils [Shumerck 1986].

En contrepartie, l'utilisation d'un squelette d'application nécessite une bonne connaissance de son fonctionnement. Elle ne supprime pas non plus la nécessité de connaître la boîte à outils sous-jacente. De ce fait, elle reste réservée aux informaticiens. Enfin, les squelettes d'application sont par

nature stéréotypés, ce qui limite les formes de dialogue et surtout le domaine d'application de chaque squelette d'application.

3.1.3 SGIU / UIMS

Un **S**ystème de **G**estion d'**I**nterface **U**tilisateur (SGIU, en anglais UIMS pour « *User Interface Management System* ») est décomposé en deux éléments : d'une part un générateur d'interface, chargé de décrire l'interface, et d'autre part un module exécutif contrôlant cette dernière lors de l'exécution. Un tel système constitue donc à la fois un outil de développement et un constructeur de dialogue [Wallace & Anderson 1993].

Du point de vue réalisation pratique, les SGIU sont des outils intéressants. Ils permettent, en effet, la création d'interfaces utilisateur avec un minimum de programmation, et, à l'inverse des catégories d'outils précédentes, ils peuvent être utilisés par un non informaticien.

Les premiers SGIU se contentaient de décrire la logique du dialogue de l'application, c'est-à-dire les réactions de l'interface aux actions de l'utilisateur[4]. La présentation et l'apparence étaient traditionnellement reléguées à l'extérieur de ces spécifications. Le dialogue était spécifié selon des méthodes variées : réseau de transition [Jacob 1986], grammaires [Olsen 1983 ; Olsen 1986] ou systèmes à événements [Singh & Green 1991]. À l'heure actuelle, on peut classer les SGIU en trois groupes : Constructeurs d'interface, Constructeurs d'interface avec langage de spécification, et Générateurs Automatiques d'interface [Duval 1994].

3.1.3.1 Constructeurs d'Interface

Les constructeurs d'interface sont des outils permettant de définir l'aspect graphique de l'interface. Ils produisent un squelette d'application auquel il faut ajouter les appels à l'application. Employant généralement la manipulation directe, ils sont d'utilisation aisée, et permettent d'obtenir très rapidement des prototypes. Cependant, ils ne permettent pas de définir la dynamique du dialogue, qui doit être codée dans l'application. Ils n'autorisent pas non plus la visualisation les éléments de l'application, à moins d'utiliser une boîte à outils pour les coder dans l'application [Meinadier 1991]. Ils permettent ainsi de dessiner aisément les différents écrans et boîtes de dialogue d'une application, mais pas de définir la succession de ces objets à l'écran, qui doit donc être codée par un développeur ayant connaissance de l'environnement.

Ce type de SGIU est bien au point, et maintenant intégré dans des produits commerciaux. Par exemple Visual Basic©, Visual C$^{++©}$ de Microsoft Corporation® ou Delphi© de Borland®, disposent de constructeurs d'interface directement intégrés à leurs environnements de programmation. Bien que limités dans leur usage, ces systèmes permettent déjà à un non-informaticien de réaliser la partie visuelle du travail de conception de l'interface.

Les systèmes universitaires en cours de développement disposent de capacités plus étendues. Myers a ainsi poussé les techniques d'inférence à un point élevé dans ce domaine avec Peridot [Myers 1988], puis GARNET [Myers et al. 1990] et GILT [Hashimoto & Myers 1992]. Ce dernier est ainsi à-même de déduire du placement *approximatif* d'un objet graphique son aspect ainsi que sa position finale.

3.1.3.2 Constructeurs d'interface avec langage de spécification

Il s'agit d'une extension des constructeurs d'interface. Dans ces systèmes, un langage spécialisé permet de définir le contrôle du dialogue, i.e. les relations dynamiques entre objets de l'interface. Les abstractions fournies permettent de faciliter la définition de ces relations sans avoir à les implémenter dans l'application. En contrepartie, il est nécessaire d'apprendre un langage de spécification.

Lorsque la méthode de description le permet, la représentation « visuelle » des spécifications est favorisée. Ceci limite les problèmes d'apprentissage du langage de spécification, notamment en

[4] Plus exactement aux événements en provenance des dispositifs d'entrée [Szekely 1996].

éliminant certains aspects ardus, comme les problèmes de syntaxe. C'est ainsi que les réseaux de transitions sont décrits par leurs schémas plutôt que par des transcriptions textuelles.

La distinction entre constructeur d'interface et constructeur d'interface avec langage de spécification n'est cependant pas aussi franche. Lorsque l'on définit un masque de dialogue à l'aide d'objets de dialogue (widgets), on fournit déjà un contrôle minimal au travers du comportement prédéterminé de ces objets. À un niveau plus élevé, Peridot [Myers 1988] permet de définir de nouveaux objets de dialogue ainsi que leurs comportements, mais ne peut définir les liens entre eux. Enfin, des systèmes comme Digis [van den Bos & Laffra 1990] [de Bruin, Bowman, & van den Bos 1993] permettent de créer l'interface complète d'une application (présentation, contrôle et interfaçage avec l'application). Il n'y a donc pas séparation nette entre les deux catégories d'outils, mais plutôt un *continuum*.

3.1.3.3 Générateurs Automatiques

L'inconvénient des deux types d'outils présentés ci-dessus est qu'ils forcent le concepteur à fournir un grand nombre de renseignements annexes du dialogue : placement, format et aspect de l'interface par exemple. Pour résoudre ce problème, certains outils génèrent une interface utilisateur à partir de spécifications de l'application. Ces spécifications sont fournies soit au moyen d'un langage spécialisé de haut niveau (exemple : UofA*UIMS, décrit dans [Singh & Green 1991]), soit directement par la liste des primitives de l'application (MIKE [Olsen 1986]). La plupart de ces outils autorisent l'utilisation d'éditeurs graphiques pour raffiner ensuite l'interface utilisateur si celle produite n'est pas satisfaisante.

Certains outils, comme ITS [Wiecha, et al. 1990], se basent sur une approche différente, où l'on définit aussi bien les spécifications de l'interface que les règles de production de cette interface. La philosophie est que la conception deviendra de plus en plus simple au fur et à mesure que le nombre de règles s'agrandira. Ceci interdit à l'inverse de modifier *a posteriori* une interface puisque le système ne peut alors comprendre pourquoi celle produite n'était pas valable.

Même si l'idée de générer automatiquement l'interface d'une application est attrayante, l'approche reste cependant confinée aux laboratoires de recherche, notamment parce que les interfaces produites par ces moyens sont généralement d'ergonomie médiocre. Le développeur doit, en outre, programmer et spécifier selon des règles strictes afin que le générateur puisse interpréter l'application.

Enfin, certains problèmes, comme la représentation de données quelconques de l'application sous une forme compréhensible pour un être humain, reste un sujet de recherche. Un générateur n'a pas connaissance de la sémantique d'une donnée (par exemple une couple d'entiers), de ce qu'elle représente dans le cadre de l'application, et peut donc difficilement représenter celle-ci de manière cohérente avec sa sémantique (un point sur un plan pour un logiciel de CAO, un couple pression/température...).

3.1.4 Les Model-Based Systems

Depuis les années 1980, les langages de spécification des IUMS ont considérablement évolué, prenant en compte de plus en plus d'éléments, que ce soit du dialogue ou de la présentation, et permettant de générer des interfaces de plus en plus riches, de plus en plus complexes. Cette évolution peut être vue comme une fusion des différentes approches employées par les UIMS.

Les systèmes actuels utilisent par exemple un modèle des tâches de l'utilisateur, un modèle des données manipulées par le système, un modèle de l'utilisateur, un modèle de la présentation… L'expression « Model Based System » ou **MBS**, fait référence à un ensemble d'outils de construction d'interface utilisant de tels modèles pour fournir une aide au développement de l'interface d'une application. Les maîtres mots sont *modèle* pour *générer* et *vérifier* le dialogue et l'interface d'une application [Szekely 1996].

L'utilisateur d'un tel système utilise un outil de modélisation pour construire le modèle général de l'application. Comme indiqué plus haut, ce modèle peut être constitué d'un ensemble de modèles particuliers (données, tâches…). Des outils automatiques de conception peuvent être employés pour passer d'un niveau conceptuel à un autre. L'emploi de ces outils peut être obligatoire ou volontaire. Des outils de validation permettent de vérifier certaines propriétés ergonomiques de l'interface. Le modèle est transformé en une représentation exécutable, qui est ensuite liée à l'application.

3.1.4.1 Modèles

Sous le terme générique de modèle se trouve le composant central des MBS. Le modèle représente les caractéristiques de l'application du point de vue de l'interface, au sens large. Ceci inclut aussi bien les notions de dialogue (ce qui est permis à un instant donné), que de présentation. Le modèle peut globalement être décomposé en trois niveaux d'abstraction [Szekely 1996] :

1. Au plus haut niveau, on trouve les modèles de tâches et de domaine. Le modèle du domaine représente les objets manipulés par l'application, leurs relations, et les actions que l'application peut effectuer sur ces objets. Le modèle de tâches représente les tâches que l'utilisateur doit pouvoir effectuer. Elles sont généralement décomposées en une hiérarchie de tâches et sous-tâches. Les tâches finales correspondent à des opérations directement réalisables avec l'application.
2. Le second niveau d'abstraction représente la structure générale de l'interface de l'application. Cette structure est décrite en termes d'interaction de bas-niveau (sélection d'un élément dans un ensemble par exemple), d'unités de présentation (correspondant aux fenêtres), et d'éléments d'information (valeur, ou ensemble de valeurs du domaine, labels, constantes…). Cette structure définit de manière abstraite les informations à présenter à l'utilisateur et les dialogues autorisés pour interagir avec ces informations.
3. Le troisième niveau forme la spécification concrète de l'interface. Elle spécifie le rendu des informations du second niveau en termes d'éléments de boîte à outils (menu, boîte à cocher, …).

Notons que la tendance actuelle est à abstraire et à expliciter les relations nécessaires entre les différents modèles [Forbrig 1999].

3.1.4.2 Outils de modélisation

Les outils de modélisation servent à assister le développeur dans la construction du modèle. Leur but est de masquer tout ou partie de la complexité du langage de modélisation. Ils assurent aussi une plus grande convivialité pour la saisie des larges quantités d'informations définissant un modèle. Enfin ils fournissent souvent une aide spécialisée à l'un ou l'autre des niveaux d'abstraction du modèle. Parmi les outils disponibles, on trouve aussi bien de (simples) éditeurs pour réaliser des spécifications textuelles du modèle (ITS [Wiecha, Bennet, & al 1989 ; Wiecha et al. 1990], Mastermind [Szekely, et al. 1995]), que des formulaires pour la création d'éléments du modèle (Meccano [Puerta 1996], Mobi-D [Puerta & Eisenstein 1998 ; Puerta 1997]) ou des éditeurs graphiques spécialisés (Humanoid [Szekely, Luo, & Neches 1993], Fuse [Lonczewski & Schreiber 1996]). Une méthodologie peut être associée aux outils, comme dans TRIDENT [Bodart, et al. 1995b].

3.1.4.3 Outils automatiques de conception

Les outils automatiques de conception permettent au développeur de spécifier uniquement certains aspects de l'interface, l'outil générant ensuite les aspects manquants à partir de ceux fournis, bien sûr, pour des classes d'applications précises et prédéfinies. Janus [Balzert 1995] [Balzert, et al. 1996], par exemple, n'a besoin que du modèle de données de l'application pour générer l'ensemble de celle-ci (interface, contrôle et noyau fonctionnel). À l'inverse, d'autres MBS, tels qu' ITS et Mastermind, ne disposent d'aucun outil automatique.

Les outils de génération automatique, lorsqu'ils sont présents, utilisent un ensemble de règles pour passer d'un modèle à un autre. Ces règles forment elles-mêmes un modèle des applications qu'il est possible de générer par ces outils. Pour reprendre l'exemple de Janus, l'outil permettant de passer

du modèle de données à l'application possède implicitement un certain modèle des tâches de l'utilisateur. Dans ce cas, il s'agit typiquement de tâches de gestion de base de données : ajout, recherche et retrait d'éléments, établissement des liens définis par le modèle de données… Le résultat de cet outil est conforme au modèle de tâches, mais ne peut générer que ce type d'application. Les outils automatiques de conception facilitent donc le travail d'un concepteur, mais limitent le domaine d'application du système.

On peut trouver dans Mobi-D [Puerta & Eisenstein 1998] une évolution de ces outils vers une aide intelligente au concepteur, en lieu et place d'une génération complète. Ainsi, U-TEL [Tam, Maulsby, & Puerta 1998] aide-t-il l'utilisateur à construire son modèle de tâches à partir d'une description textuelle de son activité. Vanderdonckt [Vanderdonckt 1999] cherche à aider l'utilisateur final, cette fois-ci dans l'optique de générer un ensemble de menus adapté.

3.1.4.4 Outils de validation

Les outils de validation sont destinés à fournir une évaluation de l'interface. Un premier exemple peut être trouvé dans les travaux sur UIDE [Foley & Sukaviriya 1994], où les modèles Keystroke et GOMS ont été utilisés pour fournir des outils automatiques d'évaluation concernant les performances des systèmes en termes d'entrée utilisateur. Une autre voie consiste à fournir des critiques et des évaluations sur les interfaces modélisées, par exemple en vérifiant la possibilité d'atteindre toutes les fonctions de l'application.

Encore peu présents dans les MBS, on peut espérer que les progrès récents enregistrés dans les approches formelles (4.1) pour les interfaces utilisateurs permettront d'incorporer plus fréquemment ce genre d'outils.

3.1.4.5 Outils d'implémentation

Les outils d'implémentation transforment les spécifications concrètes d'interface en une représentation utilisable par un programme. Celle-ci peut être directement sous forme de code source compilable (Mastermind). D'autres MBS produisent des fichiers pouvant être utilisés par des UIMS existants (FUSE). D'autres enfin ne produisent pas directement de fichier exécutable, mais fournissent un composant à compiler avec le code de l'application et permettant d'interpréter les spécifications concrètes (ITS, Humanoid).

3.1.5 Vers une conception assistée par ordinateur des applications interactives

Comme on peut le voir, les outils sont passés d'un niveau « composant élémentaire », orienté vers la présentation, à un stade de véritable « environnement de conception ». Initialement destinés uniquement aux développeurs, ils sont maintenant utilisés à tous les stades de réalisation. Ils assurent ainsi un support important aux concepteurs d'une application. Les librairies n'étaient initialement constituées que d'un ensemble de services, notamment graphiques, et ne fournissaient aucune aide pour la structuration du reste de l'application. Les générations successives d'outils ont apporté cette structure. Les outils comme les Model-Based Systems assurent ainsi aussi bien le respect de règles ergonomiques que celles du génie logiciel.

Comme les références croisées l'ont largement démontré, et même si, historiquement, la différenciation des catégories présentées dans cette section se justifie, il est patent qu'aujourd'hui, les approches se complètent largement. Ainsi, des approches boîtes à outils telle XXL intègrent-elles la notion de squelette d'application ainsi que celle de constructeur d'interface. Quant aux MBS, ils apparaissent en fait comme une synthèse des différentes approches.

Pourtant, les outils évolués ne sont, du point de vue de l'utilisateur final non informaticien, pas très séduisants. En effet, le principe des MBS est de produire un système interactif à partir de modèles déclaratifs spécifiant les aspects nécessaires à la génération. Cette dernière peut se faire par transformation manuelle assistée ou non, par génération automatique, et, beaucoup plus rarement, par interprétation. Il est souvent possible d'adapter les éléments produits comme la description graphique de l'interface, mais il est quasiment impossible alors de prendre en compte ces

modifications pour une évolution éventuelle de la conception. Les systèmes comme XXL [Lecolinet 1996], qui permet une communication entre forme graphique (et constructeur graphique d'interface) et forme textuelle de façon totalement bijective, sont encore trop rares et limités. Ceci explique peut-être l'interrogation de Szekely en 1996 [Szekely 1996] qui remarque que les développeurs n'utilisent pas actuellement de façon majoritaire les systèmes les plus évolués. Le développement le plus courant est encore aujourd'hui basé sur un squelette d'application couplé à un constructeur d'interface. Les environnements « Visual X » de Microsoft en sont le parfait exemple. Myers [Myers, Hollan, & Cruz 1996], et plus récemment Castells [Castells & Szekely 1999], pensent qu'une utilisation plus grande de la PsE/PbD serait à même d'inverser cette tendance.

Même si des conférences comme ACM UIST (User Interface Software and Technology Symposium) et surtout CADUI (Computer Aided Design of User Interfaces) cherchent à diffuser ces travaux, il est certain que les avancées actuelles de la recherche n'ont pas encore été intégrées en véritables systèmes CAO pour les interfaces utilisateurs.

3.2 Contribution

La réalisation des travaux évoqués en section 2.2 supposait de disposer au moins d'un noyau de développement d'application graphique interactive de conception technique. D'un noyau de développement à un environnement de conception, il n'y a qu'un pas. En participant aux travaux menés au laboratoire autour du modèle H^4 et des dialogues structurés (3.2.1), nous avons pu établir un modèle de dialogue original pour les systèmes de conception. Nous nous sommes ensuite attaché à en limiter les inconvénients, d'une part en nous penchant sur le retour sémantique vers l'utilisateur (3.2.2), puis en intégrant la manipulation directe aux dialogues structurés (3.2.3), étendant ainsi ces derniers. Par la suite, nous avons élaboré un outil complet de conception d'application graphique interactive utilisant les dialogues structurés étendus, incorporant la PsE/PbD, déjà partiellement présenté au chapitre 2, GIPSE. Nous présentons les différents modèles qui font de lui un véritable « Model-Based System » (3.2.4).

3.2.1 Les dialogues structurés

Dans cette première partie, nous établissons en quoi les dialogues structurés constituent un moyen naturel pour concevoir des dialogues homme-machine dans les systèmes de conception technique. Après avoir brièvement détaillé la classification des systèmes interactifs proposée au cours du Workshop DSV-IS'95 [Pierra 1995], nous montrons quelles sont les stratégies de dialogue possibles en conception technique. Enfin, nous détaillons les avantages et les inconvénients des dialogues structurés.

3.2.1.1 Classification des applications interactives

Les applications interactives possèdent des caractéristiques diverses, qui motivent des choix particuliers, tant en termes de style d'interaction ou de forme de dialogue que d'architecture logicielle. La typologie élaborée à Bonas [Pierra 1995] est une bonne base pour mettre en évidence les points particuliers d'un domaine d'application. Dans notre propos, nous allons établir les points qui définissent les applications de conception technique. Quatre critères de la typologie sont pertinents pour cette étude. Deux concernent les objets de l'application, les deux autres concernant les tâches.

- **Structure des objets**

On considère les objets du domaine comme structurés lorsque plusieurs niveaux d'objets sont accessibles par l'utilisateur. Les objets sont simples lorsqu'un objet du domaine ne peut faire partie d'un autre objet du domaine.

Les objets d'un système de conception technique sont fortement structurés. Dans ces applications, chaque objet peut correspondre à (ou faire partie de) plusieurs niveaux d'éléments : un solide est défini par une surface fermée, elle-même constituée de facettes... Lorsque l'utilisateur

sélectionne une ligne, désire-t-il sélectionner cette ligne, la surface à laquelle elle appartient, le solide qu'elle délimite [Foley, et al. 1990] ? Un système de CAO doit permettre, selon les besoins de l'utilisateur, d'exprimer l'une ou l'autre de ces interprétations.

· **Relations entre objets**

La typologie de DSVIS'95 considère les objets du domaine de l'application comme autonomes si la représentation de chaque objet du domaine est indépendante des autres objets. Les objets sont relationnels si cette représentation dépend des autres objets.

Selon ce critère, les entités manipulées par un système de conception technique peuvent être considérées comme relationnelles : l'état d'un objet dépend étroitement de l'état d'autres objets. Un nœud de maillage ne peut pas être modifié sans des vérifications importantes : vérifier d'une part que le maillage reste consistant (modèle des éléments finis), et d'autre part qu'il ne sort pas de la matière de l'objet (modèle géométrique). De la même manière, la visibilité d'un élément (facette par exemple) dépend de sa position relativement aux autres éléments du modèle. Cette visibilité ne peut être calculée par l'objet seul.

· **Arité des tâches**

Une application supporte des tâches mono-objets si chaque tâche utilisateur n'utilise qu'un seul objet du domaine de l'application. Un exemple d'application mono-objets est MacDraw™. À l'inverse, une application supporte des tâches multi-objets si les tâches peuvent comporter plusieurs objets du domaine. Une tâche multi-objets consiste par exemple à créer un cercle tangent à trois autres cercles.

Les applications de gestion utilisent des données nombreuses, mais faiblement couplées. La majorité des relations existant entre elles se situent au niveau des valeurs : si telle donnée possède telle valeur, alors telle autre donnée n'existe pas ou s'inscrit dans tel intervalle. Dans les applications de conception technique, en revanche, l'existence des relations entre objets que nous avons mentionnées précédemment induit une nature des tâches bien plus complexe.

De ce fait, les systèmes CAO permettent d'exprimer explicitement ces relations, sous forme de contraintes ou d'opérateurs géométriques entre objets, et les utilisateurs sont habitués à exprimer ces relations [Martin 1995]. Un utilisateur peut vouloir construire un cercle tangent à trois autres cercles (action multi-objets basée sur le caractère relationnel de la géométrie), ou construire un nœud de son maillage aligné verticalement avec un nœud donné, et appartenant à une frontière définie par une courbe donnée (action multi-objets basée sur la topologie d'un maillage).

· **Structure des tâches**

Une application supporte des tâches atomiques si l'utilisateur doit spécifier ses tâches indépendamment les unes des autres. Une application supporte des tâches structurées si l'utilisateur peut spécifier en pré-ordre sa hiérarchie de tâches, la tâche finale n'étant exécutée que lorsque les données ont été fournies pour toute la hiérarchie de tâches.

Dans le domaine de la conception technique, de nombreuses relations existent entre les grandeurs qui caractérisent les objets du modèle. Ces relations sont connues du concepteur, qui souhaite pouvoir les utiliser dans la construction de son modèle. Ainsi, au cours du processus de conception, le dessinateur technique connaît-il parfaitement les relations qui doivent exister entre les différents éléments de son modèle et les exploite-t-il naturellement dans sa hiérarchisation des tâches. Ces relations peuvent être numériques (la taille du segment à calculer vaut trois fois la distance entre ces deux points), géométriques (le segment à créer doit passer par le centre d'un autre objet) ou grapho-numériques (le rayon du cercle à créer vaut trois fois la distance entre deux objets) [Gardan 1991].

Les tâches doivent donc pouvoir être structurées, c'est-à-dire que l'utilisateur doit pouvoir utiliser le résultat de certaines tâches pour en accomplir une autre. Ce point est illustré par l'exemple suivant : « *Je veux construire un cercle dont le centre est le projeté de ce point sur cette ligne et dont le diamètre est égal à la distance entre ce segment et ce segment* ». Il ne s'agit pas ici de dessiner le cercle à la « bonne »

position et avec la « bonne » taille, comme on peut le faire avec des logiciels comme que MS-Draw™, mais de construire ce cercle en déclarant explicitement les relations entre objets. L'utilisateur doit (pouvoir) spécifier le centre du cercle comme étant le projeté du point sur la ligne…

3.2.1.2 Stratégies de dialogue en conception technique

La nature structurée des tâches des applications de conception technique impose un choix de structure du langage de commande. Comme tout processeur, un système interactif réalise des actions sur des objets. Deux structures sont possibles :

- Les langages postfixés, de type objet/action, sont ceux dans lesquels l'utilisateur sélectionne d'abord l'objet sur lequel agir, puis l'action à réaliser sur cet objet. Cette approche est à la base des interfaces à manipulation directe. Cette structure est adaptée aux applications offrant essentiellement des tâches atomiques. Il n'est pas possible, lorsqu'un objet est sélectionné dans un dialogue postfixé, de distinguer la sélection d'un autre objet en vue de le modifier, de l'utilisation de cet autre objet en vue de fournir une information à celui déjà sélectionné. Le choix pour les logiciels de dessin courant du type MacPaint™ ou MacDraw™ d'un langage postfixé leur interdit et d'offrir des tâches structurées, et de fournir des opérations multi-objets comme les constructions par contraintes. Les seules opérations multi-objets correspondent à une répétition de la même action sur tous les objets d'un ensemble (mais il s'agit alors d'un objet unique : l'ensemble).
- Les langages préfixés, de type commande-opérandes, sont ceux dans lesquels l'utilisateur indique d'abord l'action à réaliser puis fournit le ou les opérandes de cette action. La translation d'un objet se fait en sélectionnant la commande (bouton) « translation », en cliquant ensuite sur l'objet concerné puis en fournissant le vecteur de déplacement. On indique ici ce que l'on va faire, puis comment on va le faire. Cette structure commande-opérandes est considérée comme la seule utilisable lorsque des tâches structurées sont nécessaires [Guittet 1995].

La nature des tâches des applications de conception technique impose donc des stratégies de dialogue particulières afin d'autoriser les tâches structurées et multi-objets. C'est ce que nous discutons ci-dessous.

La conception technique est un domaine où les règles de construction jouent un rôle fondamental. À l'inverse des systèmes de dessin grand public, où le positionnement des objets peut-être approximatif, les systèmes de conception technique fournissent à leurs utilisateurs des modes de construction d'objets par contraintes. Là où un logiciel de la première catégorie permettra de créer une ellipse par deux clics souris, puis de l'agrandir et de la déformer en cercle par manipulation directe, un système de conception technique fournira des primitives distinctes de création d'ellipses et de cercles, basées sur leurs caractéristiques géométriques (centre, rayons). Ce qui pourrait paraître frustre au premier abord (création d'un cercle par centre et rayon) s'avère en fait très puissant, tout en restant parfaitement défini *a priori*, par l'usage intensif des tâches structurées. Ainsi peut-on créer, dans les systèmes de conception technique, un cercle par deux points (centre et rayon), eux-mêmes calculés par des sous-tâches très complexes.

Prenons l'exemple de la construction suivante :

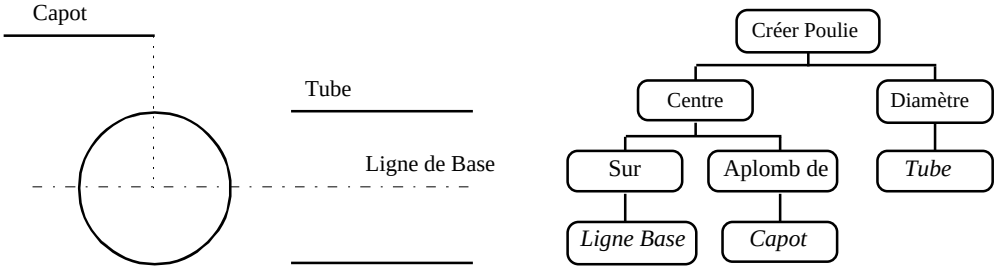

Figure 5 : Exemple de construction mécanique, et son arbre de tâches associé

Il s'agit d'une partie de mécanisme où le cercle (vue en deux dimensions d'une poulie) doit pouvoir s'ajuster dans le tube de droite (symbolisé par les deux segments horizontaux), et est positionné au départ à l'aplomb d'un capot (l'élément supérieur). Le but du concepteur est donc le suivant : « *Création d'une poulie positionnée sur la ligne de base à l'aplomb de l'extrémité du capot, et dont le diamètre est égal au diamètre intérieur du tube* ». Trois grandes approches peuvent être utilisées dans les systèmes de conception technique pour résoudre ce problème.

· **La méthode « table à dessin »**

Le travail classique d'un concepteur sur une table à dessin consiste à bouleverser son arbre de buts/sous-buts. Il commence par identifier les entités supports de la construction désirée (droites, cercle, points…). Les entités constituant la construction finale (segments, arcs de cercle…) sont ensuite ajoutés en s'appuyant sur ces entités supports. La stratégie dite « table à dessin » consiste à plaquer cette méthode de travail sur le support informatique. Une application employant cette stratégie de dialogue dispose donc d'un très grand nombre de fonctions de création d'entités.

Dans notre exemple, il faut donc commencer par créer la verticale passant par l'extrémité du capot, puis la ligne de base et enfin le point qui est leur intersection, centre du futur cercle. Ensuite, il faut prolonger un côté du tube et créer le cercle centré sur le point et tangent au côté prolongé. Enfin, il est nécessaire de détruire (ou masquer) les éléments de construction.

Cette approche, très classique, est disponible sur la quasi-totalité des systèmes existants. Elle respecte les habitudes des utilisateurs (ceux qui sont passés par la table à dessin), mais nécessite la mise en place d'une stratégie très différente de la simple hiérarchie buts/sous-buts initiale. Elle nécessite l'ajout de tâches intermédiaires et de tâches finales (comme celle de nettoyage). L'écho sémantique est correctement effectué pour chacune des sous-tâches intermédiaires, mais aucun écho sémantique de la tâche globale n'est visible avant la fin de la construction. Toute erreur dans la réalisation d'une sous-tâche oblige la révision globale de la stratégie, et ce uniquement à la fin de la tâche globale.

De plus, l'utilisation d'une stratégie identique à celle de la table à dessin fait souvent abstraction des potentiels des systèmes informatiques, et conduit à des stratégies de dessin sous-optimales [Bhavnami & John 1996]. Ces auteurs citent, par exemple, des surfaces fermées construites sous forme d'un assemblage de segments indépendants, ce qui interdit leur remplissage automatique par un motif, que l'utilisateur est donc obligé de dessiner lui même. Ce qui est ici en cause n'est pas la complexité du système de conception, généralement maîtrisé par l'utilisateur, mais bien la décomposition de la tâche par cet utilisateur.

· **La méthode à « post-contraintes »**

L'augmentation de puissance des systèmes informatiques a permis de mettre en œuvre des techniques beaucoup plus séduisantes. L'idée générale consiste à dessiner approximativement, par exemple avec des techniques de manipulation directe, puis à contraindre *a posteriori* le schéma obtenu. Le système est ensuite en mesure d'ajuster le dessin aux contraintes exprimées. Dans notre exemple, les tâches se décomposent en un dessin approximatif du cercle, puis en l'expression des trois contraintes (alignement vertical par rapport à l'extrémité du capot, alignement par rapport à l'axe du tube, et même rayon que ce dernier).

Cette méthode, qui peut sembler idéale, présente néanmoins quelques limites : elle n'est pas si simple à utiliser, car les contraintes exprimées peuvent être insuffisantes (problème sous-contraint, impossible à résoudre) ou trop nombreuses (problème sur-contraint, avec possibilités de contradictions) ; de plus, elle est circonscrite à certains types de problèmes (2D par exemple). Nonobstant ces limites, elle est présente dans de nombreux systèmes (I-Deas, Catia, Pro-Engineer...), dans le module de dessin 2D fréquemment appelé « sketcher ».

On remarquera cependant qu'elle nécessite une transformation complète de l'arbre des buts/sous-buts, et qu'elle cache la méthode de construction qui devient un peu « magique ». De plus, l'approximation de départ faite par l'utilisateur peut, en fait, se révéler totalement remise en cause

par la résolution du système de contraintes. Il n'est ainsi pas rare de s'apercevoir que, par un mauvais choix de contraintes, le dessin calculé n'a pas de rapport avec la solution recherchée. La notion d'écho sémantique est ici totalement absente.

- **La méthode par « dialogues structurés »**

Le principe de base des dialogues structurés consiste à fournir à l'utilisateur le moyen de calquer au plus près son arbre des tâches exprimées sur son arbre de buts/sous-buts. Ceci se fait au moyen d'un dialogue structuré, permettant d'exprimer les paramètres des tâches[5] au moyen d'autres tâches. Ainsi, dans notre exemple, l'arbre des tâches deviendra-t-il la création d'un cercle dont le centre est la projection sur la ligne de base de l'extrémité du segment représentant le capot, et dont le rayon est égal à la distance séparant le haut du tube de la ligne de base (rayon du tube).

Cette forme de dialogue peut être vue comme un ensemble de producteurs et de consommateurs : certaines tâches du système *consomment* des données *produites* par d'autres tâches. Par exemple, la tâche de création de cercle reçoit une partie de ses données de la tâche de calcul de la projection d'un point sur un objet. Ces tâches productrices consomment elles-mêmes des données, en provenance soit d'autres producteurs, soit de l'utilisateur. Celui-ci agit comme le producteur de données initial.

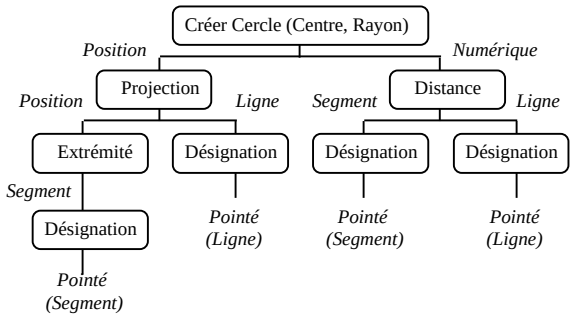

Figure 6 : Arbre de tâches du système (les paramètres sont en italique)

La mise en œuvre de cette méthode nécessite, pour éviter l'explosion combinatoire des agencements de sous-tâches, le développement d'une architecture d'application particulière, comme celle décrite dans [Guittet 1995], [Gardan, et al. 1988 ; Gardan, Jung, & Martin 1993 ; Martin 1995], ou [Qiang, et al. 1997]. Elle est utilisée de manière ponctuelle dans quelques systèmes (I-Deas, Catia, Pro-Engineer, Autocad...).

La plupart des systèmes de CAO commerciaux n'emploient pas qu'une seule des méthodes présentées ci-dessus. Beaucoup conjuguent la méthode « table à dessin » avec l'une des deux autres. Cette stratégie est, en effet, la plus simple du point de vue implémentation logicielle, les fonctions n'étant pas liées les unes aux autres comme dans les dialogues structurés, ou aussi complexes que la résolution des équations associées aux dialogues à post-contraintes. Par exemple, Autocad© Version 12 emploie majoritairement la méthode « table à dessin », mais autorise pour certaines fonctions l'emploi de « fonctions d'accroche » pour fournir les points particuliers d'objets (se plaçant alors dans les logiciels utilisant les dialogues structurés). D'autres systèmes tels que Pro-engineer© utilisent « table à dessin » et « post-contraintes ».

3.2.1.3 Mise en œuvre des Dialogues Structurés

La mise en œuvre des dialogues structurés nécessite de distinguer au sein du contrôleur de dialogue deux types de tâches : les tâches terminales et les sous-tâches d'expression. Les premières constituent les actions principales de chaque phrase du dialogue ; elles modifient généralement le modèle du noyau fonctionnel (*Créer point*, *Créer cercle*), mais peuvent également toucher la

[5] Nous employons tout au long de cette section le terme de « tâche » dans un sens correspondant à des tâches de niveau relativement bas, telles que « créer cercle » ou « calculer intersection »… Ce sont les tâches du système, par opposition aux tâches de l'utilisateur du genre « créer un mécanisme qui… »

présentation de l'application (*zoom, changement de point de vue, etc.*). À l'inverse, les sous-tâches d'expression ont pour but de calculer des expressions, afin de mettre le résultat à la disposition d'autres (sous-)tâches. Ces expressions peuvent être de simples accès à des valeurs caractéristiques (Centre de, Extrémité de, etc.), ou bien constituer des expressions géométriques complexes (*Projection de, distance, etc.*). Ces sous-tâches de production effectuent une transformation de leurs paramètres d'entrée en paramètres de sortie qui sont eux-mêmes utilisés comme paramètres d'entrée dans la tâche en cours.

Laurent Guittet [Girard, Pierra, & Guittet 1995 ; Guittet 1995 ; Guittet & Pierra 1993] a ainsi proposé de définir un système interactif comme un ensemble de cinq sous-systèmes, dont quatre sont structurés sous forme d'une hiérarchie d'agents. Ces quatre hiérarchies parallèles contiennent des agents différents structurés selon des critères différents, et constituent l'essence du modèle H^4. Les dialogues structurés s'implémentent très facilement au sein de H^4 sous la forme d'interacteurs hiérarchisés. Nous discuterons ce point plus en détail à la section 4.2.1.

Les dialogues structurés présentent cependant un inconvénient. S'ils permettent d'exprimer des tâches très complexes, ils doivent également en subir la contrepartie : ils « perdent » souvent l'utilisateur. En effet, il n'est pas aisé de réaliser un retour d'information sémantique dans ce style de dialogue. De plus, ils sont lourds à utiliser, et ne se justifient pas dans toutes les situations de dialogue. Ce sont ces deux points que nous évoquons dans les sections suivantes.

3.2.2 Exploration de la tâche en cours

Les travaux de Norman [Norman 1986] ont défini un modèle largement accepté aujourd'hui de l'activité humaine face à l'ordinateur. Le raisonnement en buts/sous-buts permet de décomposer un problème complexe en sous-problèmes plus simples. L'application récursive de ce type de raisonnement permet d'arriver au niveau des tâches élémentaires, dont l'exécution est soumise au cycle perception/action. Au cours de ce dernier, le système permet à l'utilisateur de transformer ses objectifs en actions dont les effets lui sont rendus perceptibles afin de lui permettre d'évaluer le plus tôt possible leur adéquation avec le but visé. Les dialogues structurés permettent à l'utilisateur de transformer directemenr ses buts/sous-buts en une succession de commandes du système.

Cependant, la perception correcte et précoce des effets des actions est l'un des points majeurs permettant à l'utilisateur d'éviter de se perdre dans la forêt de ses sous-buts [Hix & Hartson 1993 ; Preece, et al. 1994]. La seule rétroaction ne permet pas d'éviter les erreurs de l'utilisateur, ce qui oblige le plus souvent l'incorporation de mécanismes « Défaire / Refaire » (UNDO/REDO en anglais) dans les systèmes complexes.

Afin d'améliorer la prévention des erreurs, des mécanismes d'échos proactifs (« proactive feedback » en anglais) peuvent être implémentés. Les menus grisés ou les fantômes de la manipulation directe en sont deux exemples qui répondent à deux grands besoins : la prévention automatique et la prévention informative. Les menus grisés sont la représentation visuelle de l'interdiction faite à l'utilisateur d'utiliser des actions non appropriées à un moment donné. Il s'agit d'une *prévention automatique* (et contraignante). À l'inverse, les fantômes de la manipulation directe se contentent de montrer, tout au long de l'action en cours (« Glisser / Déplacer » ou DRAG & DROP), *ce qui se passerait si* l'utilisateur terminait son action : le fichier serait déplacé *ici*, ou bien serait mis dans *la corbeille qui est actuellement mise en évidence*. Cet écho par anticipation joue un rôle d'information auprès de l'utilisateur en lui donnant une idée du résultat de l'action avant son exécution, ce que l'on peut qualifier de *prévention informative*.

On remarquera, que dans les deux cas, il ne s'agit que de conventions de dialogue, pas nécessairement en rapport avec l'accomplissement réel de la tâche en cours : ainsi, l'écho du passage d'une icone sur une fenêtre représentant un dossier ouvert est-il la mise en évidence de la fenêtre, et non pas l'insertion simulée du fichier manipulé (ou de son fantôme) à l'endroit où il serait effectivement mis. Cette remarque conduit à la conclusion que ces échos proactifs sont des techniques d'interaction propres au dialogue, et dont la partie sémantique est limitée et conventionnelle. Leur introduction dans un dialogue pré-existant demande une profonde

modification de ce dernier. Il est pourtant clairement établi que ces échos permettent d'améliorer sensiblement la qualité ergonomique des applications interactives [Hix & Hartson 1993].

Si l'utilisation de dialogues structurés présente l'avantage de diminuer très fortement la distance sémantique (l'arbre des tâches est très proche de l'arbre des buts/sous-buts), évaluer où l'on se situe en cours d'interaction n'est pas aisé lorsque le dialogue se complexifie. Ainsi, lorsque l'on doit effectuer sept commandes et quatre désignations d'objets à l'écran comme dans la tâche décrite précédemment, il est fréquent de perdre le fil de sa démarche, ce qui engendre des erreurs d'exécution de la tâche. En effet, l'écho fourni par ces systèmes se limite à un écho articulatoire (mise en évidence du dernier élément sélectionné) pendant les sous-tâches, éventuellement par un écho de la sous-tâche en cours de plus bas niveau, et par un écho sémantique final de la tâche principale (construction du cercle, lorsqu'il est complètement défini, dans notre exemple), bien tardif et non proactif. Ainsi, la conscience de la situation dans l'arbre des tâches n'est pas évidente, par manque d'écho sémantique, l'écho des sous-tâches est inexistant, et les erreurs ne sont découvertes qu'à la fin de la tâche globale.

L'objectif de notre contribution a été de proposer un moyen simple et systématique pour introduire un écho proactif au sein d'une application supportant un dialogue structuré. La solution mise en œuvre permet de réaliser un système où toute tâche dispose d'un retour d'information sémantique lors de sa réalisation, et ce, quel que soit le nombre de sous-tâches entrant en jeu. En outre, cet écho étant géré au niveau de chaque (sous-) tâche, on dispose d'un écho sémantique multi-niveau proactif.

3.2.2.1 Écho sémantique et dialogues structurés

Le premier moyen d'obtenir un écho sémantique est d'incorporer de façon systématique dans le dialogue structuré la technique dite de Rubber Banding [Foley et al. 1990], que l'on peut traduire en français par « segment élastique ». Cette technique consiste à visualiser le résultat d'une action alors que tous ses paramètres ne sont pas complètement fournis. Il s'agit en d'autres termes de présenter à l'utilisateur le résultat potentiel de son action courante. L'exemple le plus simple est celui de la construction d'un segment, où l'on dessine un « segment élastique » entre une première position et la position courante de la souris, bouton enfoncé. Le segment définitif est créé à partir de la position de la souris lorsque l'utilisateur relâche le bouton.

Dans le cadre des tâches structurées, ceci n'est pas suffisant pour fournir un écho de la tâche complète. En effet, la position du second point du segment peut n'avoir qu'un lointain rapport avec la position de la souris, et être déterminée par une expression complexe mettant en jeu des sous-tâches de production comme la projection ou l'intersection. Pour obtenir un bon écho sémantique de la tâche globale, il conviendra de prendre en compte cette partie de l'interaction et de présenter un écho de chacune des (sous-)tâches en jeu. L'ensemble de ces échos forme ce que l'on nomme l'exploration de la tâche finale.

Qu'elle utilise un dialogue sous forme simple ou structurée, incorporer un écho sémantique au sein d'une application introduit un certain nombre de problèmes : constance du modèle et gestion de l'affichage, en particulier. En outre, d'autres particularités spécifiques aux dialogues structurés apparaissent lors de cette introduction, comme le problème de la cohérence du dialogue ou celui du contrôle de la fin de tâche [Amai 1994].

- **Constance du modèle**

L'incorporation d'un écho dans une fonction est globalement réalisé de la même manière, que cette fonction appartienne à un dialogue structuré ou non. Dans un premier temps la fonction acquiert, de manière cyclique, des informations temporaires (« fugitives »), généralement acquise de façon pseudo-continue lorsque l'utilisateur bouge la souris avec un bouton enfoncé. La fonction fournit, à partir de ces valeurs, un écho correspondant au résultat potentiel de l'action si ces valeurs étaient employées (« écho fugitif »). Dans cette phase, le modèle géométrique n'est pas modifié. Dans une seconde phase, la valeur définitive est acquise. C'est à ce moment que le modèle est modifié pour y incorporer le nouvel objet.

Dans un dialogue utilisant des tâches multi-objets, une tâche ne pouvant s'exécuter que lorsque tous ses paramètres lui sont fournis, un comportement fugitif ne peut se manifester que par rapport au dernier paramètre, qui seul peut être fugitif. Si le dialogue permet à l'utilisateur de choisir l'ordre des paramètres fournis à une tâche, le paramètre fugitif est le dernier dans l'ordre chronologique.

- **Gestion de l'affichage**

La présence d'un écho dynamique au sein d'une fonction implique qu'à chaque acquisition d'une donnée fugitive, cette fonction doive, avant d'afficher l'écho du résultat courant, effacer l'écho associé à la valeur précédente, s'il existe.

Ceci peut être réalisé de plusieurs manières : la fonction peut, par exemple, disposer d'un état et d'une mémoire associée lui permettant de savoir si elle a déjà produit un écho, et les caractéristiques de cet écho. Cette solution, simple à mettre en œuvre, est relativement lourde en ce sens que toute fonction doit alors disposer d'un tel couple (état, mémoire). En outre, dans le cas d'un dialogue structuré, une même fonction ne peut être employée simultanément qu'à un seul niveau de l'arbre des tâches. Une autre solution consiste à fournir au développeur du système un composant spécialisé, auquel le contrôleur de dialogue a accès, pour réaliser les échos fugitifs. C'est alors ce contrôleur de dialogue qui prendra en charge l'effacement des échos résultant des appels précédents à la fonction.

Les deux points ci-dessus concernent tous les dialogues. Les points suivants sont spécifiques aux dialogues structurés.

- **Cohérence du dialogue**

L'exploration au sein d'un dialogue structuré peut être vue comme la pseudo-exécution de la fonction terminale : tous les paramètres sont fournis, mais seul l'écho de l'action est réalisé. Du point de vue du dialogue, ceci suppose que ce dernier n'évolue pas. Cette règle doit être étendue à toute la hiérarchie des sous-tâches en cours. Un écho fugitif comportant un nombre quelconque d'appels (jusqu'à ce que l'utilisateur relâche la souris par exemple), les paramètres non fugitifs doivent être conservés entre chaque appel de la fonction fugitive, et ce jusqu'à l'appel final. D'un point de vue plus général, c'est tout l'état du dialogue qui est concerné. L'appel d'une fonction avec des paramètres fugitifs ne doit pas modifier le dialogue, ce qui implique également qu'une fonction utilisant une donnée fugitive ne doit pas consommer (détruire) les paramètres qui lui sont transmis. Ceux-ci doivent rester valides après appel de cette fonction.

- **Transmission du caractère fugitif**

Dans les dialogues structurés, les paramètres sont transmis de tâche en tâche, et interprétés au sein de différentes fonctions. Si une fonction recevant un paramètre fugitif retournait un paramètre non fugitif, la tâche de niveau supérieur en attente de ce dernier exécuterait sa fonction définitive, alors que l'utilisateur n'a pas terminé son interaction. Il est donc important de ne pas perdre le caractère fugitif des données lors des transformations. Tant que l'utilisateur émet des données (fugitives), toutes les tâches actives doivent recevoir des paramètres fugitifs, quelle que soit leur position dans cette hiérarchie. Tout paramètre produit par une fonction à l'aide de paramètres fugitifs est dont lui même fugitif. Ceci peut être automatiquement assuré par le contrôleur de dialogue.

- **Persistance de l'écho et contrôle de la fin de tâche**

Au sein d'un dialogue non structuré, il n'existe qu'une seule fonction active à un instant donné, et donc qu'un seul écho. En outre cet écho n'apparaît que lorsque le dernier paramètre est en cours de saisie. Au contraire, au sein d'un dialogue structuré, plusieurs fonctions sont actives à un instant donné, et peuvent chacune fournir leur écho.

En outre, ces échos sont disponibles bien avant le dernier paramètre de la tâche terminale, à savoir les échos des sous-tâches de production ayant fourni les premiers paramètres de cette tâche.

Ces échos doivent rester visualisés même après la fin de la sous-tâche de production qui les a produits, et ce afin de fournir à l'utilisateur un rappel visuel des informations déjà transmises à la tâche finale. Ces échos doivent perdurer jusqu'à la fin de la cette dernière, moment où ils doivent disparaître. Il convient donc, pour un retour d'information maximal vers l'utilisateur, de contrôler l'effacement des échos présents à l'écran lorsque s'achève la tâche terminale.

3.2.2.2 Réalisation

Les différentes solutions proposées pour résoudre les problèmes décrits ci-dessus ont été validées par la réalisation d'un logiciel de conception technique. Tel qu'il a été réalisé, le système comporte plusieurs avantages. En premier lieu, et conformément au but qui lui était assigné, il permet d'assurer une exploration d'un dialogue structuré, et ce, quelle que soit la profondeur de structuration arborescente de ce dialogue.

En second lieu, le générateur de dialogue développé pour fabriquer le système permet une évolution facile d'une application de conception technique usuelle vers une application comportant l'exploration. Seules sont à ajouter des actions pré-visualisant le résultat des actions pouvant modifier le modèle. Les actions d'expression sont directement réutilisables, sans aucune modification du source, et avec un minimum de modification du dialogue.

L'ensemble des problèmes exposés initialement n'est cependant pas complètement résolu. En effet, si l'utilisateur peut avoir une meilleure appréciation de la tâche en cours, il ne peut revenir en arrière que durant une même arborescence de sous-tâche. Il peut modifier l'entrée en cours, mais non celles qu'il a déjà réalisées. L'amélioration de ces différents points fait l'objet d'un travail en cours.

3.2.3 Dialogues structurés et manipulation directe

Dans la précédente section, nous avons présenté notre contribution à la réalisation d'un retour d'information proactif des actions de l'utilisateur sur le système. L'introduction de ce retour d'information dans les dialogues structurés était destinée à limiter le gouffre d'évaluation auquel était confronté l'utilisateur lors de phases de dialogue complexes.

Le but de la présente section est de présenter les travaux réalisés pour s'attaquer à l'autre gouffre défini par Norman, le gouffre d'expression. Il s'agit de simplifier pour l'utilisateur l'expression du but qu'il souhaite atteindre. Plusieurs stratégies existent pour atteindre cet objectif. Ainsi, certains systèmes commerciaux de conception technique utilisent-ils une forme particulière d'exploration pour faciliter l'expression de relations entre objets. Il s'agit pour ces systèmes de simplifier l'expression de tâches complexes.

Une deuxième approche consiste à autoriser l'utilisateur à employer une autre forme de dialogue lorsque la stratégie courante employée au sein de l'application est trop lourde vis-à-vis de la tâche à accomplir. Ceci revient à intégrer à l'application différentes formes de dialogues. Certains styles d'interaction sont plus adaptés que d'autres aux dialogues simples que l'interaction sous forme de dialogues structurés. C'est notamment le cas de la manipulation directe.

3.2.3.1 Intérêt de l'intégration

La manipulation directe est basée sur une forme de dialogue simplifiée, où l'on agit principalement sur un objet à la fois, et où les relations entre objets ne peuvent guère être exploitées. En tant que telle, cette méthode de dialogue semble peu adaptée au domaine qui nous intéresse principalement, où les objets sont souvent créés et modifiés à partir d'attributs d'autres objets, ou de relations entre ces objets.

Cependant, même dans ces applications basées sur les tâches multi-objets, il existe des moments où l'utilisateur souhaite réaliser des actions simples, mono-objets. Par exemple, lors de la préparation d'un plan, plusieurs vues d'un même objet sont représentées sur un même calque. Le placement de chacune de ces vues est indépendant des autres. L'utilisateur peut vouloir déplacer de manière générale l'une de ces vues, sans pour autant utiliser les fonctions de translation par vecteur.

Cette dernière, dans un système CAO classique, nécessite d'une part d'invoquer la commande associée, d'autre part de désigner l'objet et enfin de donner deux positions formant le vecteur de translation. Dans ce cas, le dialogue est trop lourd comparé à la simplicité de l'action que souhaite réaliser l'utilisateur. Il agit alors comme un frein plus que comme une aide.

Soulignons qu'à l'inverse, il existe de nombreuses situations où la manipulation directe atteint ses limites. Celle-ci repose en effet sur les actions physiques de l'utilisateur, ce qui implique un problème de précision. Lorsqu'une plaque doit être placée à exactement 10,4 cm d'une autre plaque, avec une tolérance de 0,1 mm, il devient difficile de réaliser ce placement « à la main ». Les outils couramment disponibles dans les applications à manipulation directe que sont le zoom et la règle n'éliminent pas ce problème de précision : il n'est pas rare, après avoir placé un objet via un zoom, de découvrir que celui-ci n'apparaît pas au bon endroit sous un autre facteur de zoom. Mieux vaut alors utiliser les fonctions du dialogue structuré. Ces dernières fournissent une précision absolue dans le placement, là où la manipulation directe permet une précision relative.

L'intérêt de la manipulation directe parait alors non pas de remplacer les formes structurées de dialogue, mais de suppléer ces formes dans les cas où elle est plus efficace. Il s'agit de fournir à l'utilisateur des moyens alternatifs pour réaliser certaines actions. Pour reprendre l'exemple de la translation, on ne souhaite pas remplacer la méthode structurée, laquelle est toujours utile lorsqu'on souhaite une translation en fonction d'autres paramètres. On veut donner à l'utilisateur la possibilité de déplacer directement l'objet à l'écran lorsque cela lui convient mieux.

De plus, une réelle intégration de la manipulation directe au sein d'un contrôleur de dialogue structuré permet d'obtenir des comportements inaccessibles à un seul noyau de manipulation directe. Par exemple, le changement d'attributs d'un objet, comme la couleur par exemple, est un problème peu facile à résoudre en manipulation directe pure. La méthode indirecte classique consiste à fournir une boîte de dialogue où l'utilisateur pourra sélectionner la couleur désirée. Cependant, il ne lui sera pas possible par ce biais de fournir « la même couleur que cet objet *là* »… Beaucoup de systèmes graphiques résolvent le problème par l'utilisation d'une « pipette » supposée « aspirer » la couleur d'un objet désigné puis la transmettre à l'objet dont on veut changer la couleur. Cette méthode s'apparente à celle du « couper/coller » qui passe par un tampon externe invisible, rompant ainsi avec un des axiomes de la manipulation directe, qui consiste à dire que les actions doivent porter sur des objets concrets. L'intégration complète de la manipulation directe au sein d'un système à dialogues structurés permet par exemple de fournir comme couleur à un objet (celui actif en manipulation directe) la couleur interpolée entre deux autres objets (sous-tâche de production du dialogue structuré).

3.2.3.2 Réalisation

La description des solutions que nous avons proposée pour répondre aux besoins évoqués ci-dessus introduit de nombreux éléments techniques qui sortent largement du cadre de ce mémoire. Ils peuvent être trouvés dans [Patry 1999a]. Les différentes solutions d'intégration des deux systèmes de dialogue y sont étudiées, et les services minimaux à fournir, tant du côté du contrôleur de dialogue que de celui du noyau fonctionnel y sont recensés.

Du point de vue du développeur, le coût de ces nouvelles possibilités est minimal. Le contrôle du dialogue est entièrement assuré par le système. Le développeur doit fournir deux composants : d'une part les fonctions employées par la manipulation directe, et d'autre part la description de l'utilisation de ces fonctions, sous forme d'un modèle du dialogue de manipulation directe. En ce qui concerne les fonctions, il faut noter qu'une grande partie de celles-ci existe déjà dans le système. Seule leur utilisation au sein du dialogue est différente.

L'introduction de la manipulation directe au sein d'un composant basé initialement sur les dialogues structurés permet à l'utilisateur d'employer, de manière transparente pour lui, la stratégie de dialogue qui lui semble la plus adaptée à la tâche en cours. Elle permet à un utilisateur de disposer à la fois de la puissance d'expression des dialogues structurés et de la facilité d'utilisation de la manipulation directe, selon ses besoins.

3.2.4 GIPSE : un « Model-Based System »

GIPSE, présenté sommairement à la section 2.2.4.3, est un système permettant, à partir d'un noyau fonctionnel existant, de définir une application supportant des dialogues structurés. GIPSE s'appuie sur une description sous forme de modèle du dialogue de l'application pour générer et gérer ce dernier. Nous présentons dans un premier temps les différents modèles qui lui sont nécessaires (3.2.4.1) ; dans un deuxième temps, nous voyons comment GIPSE utilise la couche graphique pour définir la présentation et les relations avec le modèle (3.2.4.2). Enfin, nous examinons en quoi GIPSE constitue un véritable environnement de CAO de systèmes interactifs (3.2.4.3).

3.2.4.1 Les modèles de GIPSE

Le concepteur d'une application doit fournir un ensemble de descriptions, ou modèles, décrivant les propriétés de la future application. Ces modèles sont au nombre de quatre :

- Modèle des objets, qui définit l'ensemble des objets manipulables par l'utilisateur.
- Modèle des tâches, qui définit chaque tâche indépendamment les unes des autres. C'est à ce niveau que l'on va trouver les informations comme la nature et le sens des paramètres d'une tâche, l'aide en ligne…
- Modèle du dialogue, qui définit les contraintes statiques du dialogue, comme la succession de tâches autorisées. Ce modèle donne à la fois la hiérarchie des interacteurs vue par le moniteur et la structure interne de chaque interacteur. Deux cas peuvent se présenter, selon que l'on souhaite ou non intégrer la manipulation directe au système.
- Modèle de la présentation, indiquant principalement l'agencement des menus.

L'ensemble des modèles est donné au système sous forme de fichiers d'initialisation ASCII. Il existe un fichier pour chaque modèle. En annexe, est fournie une publication qui présente quelques exemples des notations utilisées.

- **Le modèle des objets**

Les objets manipulés par une application conçue avec GIPSE dépendent étroitement du noyau fonctionnel. Mais il n'est pas question dans GIPSE de décrire dans l'interface l'ensemble des relations gouvernant les objets du noyau fonctionnel. La description des objets de l'application se limite donc à une simple structuration pour les seuls besoins du dialogue.

Ainsi, décrit-on la liste des objets de base manipulables (jetons), qui doivent correspondre aux types d'objets manipulés par le noyau fonctionnel, ainsi que l'association de ces objets pour former des groupes (ou unions). Ces groupes permettent de factoriser en une seule un ensemble de fonctions du système. Une union peut être construite à partir d'un ensemble de jetons ou d'unions. Cette description sous forme de jetons et d'unions donne une hiérarchie.

- **Le modèle des tâches**

Ce modèle décrit l'ensemble des tâches du système que l'utilisateur peut déclencher. Il permet de passer des tâches de l'utilisateur aux fonctions du système. Il permet aussi d'associer aux tâches des informations sémantiques qui seront présentées à l'utilisateur pour faciliter sa compréhension du système (ligne d'aide, paramètres par exemple).

Pour chaque tâche, le concepteur indique la spécification de la fonction encapsulée du système, sous forme d'un nom, des paramètres attendus par cette fonction, et de l'éventuel paramètre de retour (dans le cas d'une tâche de production). Cette spécification doit correspondre à la spécification d'une fonction implémentée dans le système. Une tâche terminale possède deux fonctions : la première indiquée correspond à la fonction finale, appelée lorsque aucun paramètre n'est fugitif, tandis que la seconde fonction est celle utilisée lorsqu'un de ces paramètres est fugitif (pour l'exploration des tâches terminales).

Outre les fonctions les tâches possèdent aussi, de manière optionnelle, une ligne d'aide qui apparaît en bas de l'écran lorsque le curseur se place sur un bouton déclenchant la tâche. Enfin, les

tâches peuvent avoir pour chaque paramètre de leur fonction une sémantique, c'est-à-dire un texte associé au paramètre.

- **Le modèle de dialogue**

Le modèle du dialogue définit les informations sur les structures statique et dynamique du dialogue. Celui-ci est décomposé en un ensemble d'interacteurs nommés. Les interacteurs correspondant chacun à un « niveau » de tâche de l'utilisateur, la structure du dialogue associée à l'un d'entre eux correspond à la structure du dialogue pour le niveau de tâche correspondant.

L'ordre d'apparition des interacteurs définit leur ordre dans le moniteur : les interacteurs reçoivent leurs données du moniteur dans l'ordre inverse de leur définition. Si l'on définit l'interacteur A puis l'interacteur B, toute entrée de l'utilisateur sera présentée automatiquement à B avant d'être (éventuellement) transmise à A.

Le modèle de dialogue se décompose en deux sous-modèles, l'un concernant les dialogues structurés standard, et autorisant les interactions multi-objets, et l'autre réservé à la manipulation directe, et n'autorisant ainsi que des actions mono-objet.

- **Dialogue standard multi-objet**

Chaque interacteur est défini au moyen d'un ensemble hiérarchique d'opérateurs de dialogue. Ces opérateurs de dialogue correspondent aux opérateurs de composition de SACADO [Gardan et al. 1988 ; Gardan, Jung, & Martin 1993], ou aux constructeurs de MAD [Scapin & Pierret-Golbreich 1989 ; Scapin & Pierret-Golbreich 1990].

Ces opérateurs sont de quatre sortes : opérateurs d'alternative, de séquence, de séquence ininterruptible et d'exécution. Tous les opérateurs peuvent être dans trois états : **Désactivé**, lorsque la portion du dialogue où se situe l'opérateur n'est pas accessible à l'utilisateur, **Activé** lorsque cette portion est accessible, et **Validé** lorsque l'opérateur cesse d'être accessible. Généralement, la validation d'un opérateur entraîne l'activation d'un autre opérateur.

Grâce à la structure sous forme d'interacteurs hiérarchisés, ces quatre opérateurs seulement sont suffisants pour décrire les formes de dialogue nécessaire. L'adjonction d'une opération « répétition infinie » permet de compléter la définition. Grâce à la structuration des jetons, on peut atteindre un niveau de concision extrêmement important.

- **Modèle pour la manipulation directe**

De la même manière que pour les interacteurs de dialogue structurés, l'interacteur de manipulation directe est dynamiquement créé par le biais d'une description de plus haut niveau du dialogue qu'il encapsule. Cette description, qui modélise le dialogue de manipulation directe, est regroupée au sein d'un fichier d'initialisation qui définit les réactions de chaque type d'objet en fonction des données que transmet l'utilisateur. Une réaction correspond à une fonction du noyau fonctionnel, appelée lorsque l'objet reçoit une donnée de type particulier et qu'il se trouve dans un mode donné.

Ces réactions sont, pour chaque type d'objet, classées en trois groupes : réactions vis-à-vis des actions directes de l'utilisateur sur l'objet, réactions vis-à-vis des actions de l'utilisateur sur les poignées de l'objet, et enfin réactions vis-à-vis des valeurs que l'on souhaite transmettre à l'objet. À chaque type de donnée est associée une fonction de traitement par défaut. Il est possible de rajouter des nouvelles fonctions associées à un type en joignant un mode.

Ces différents modes forment les commandes spécifiques de l'interacteur de manipulation directe, et sont accessibles par un menu contextuel créé automatiquement par le système à l'initialisation du dialogue (clic droit du bouton de la souris). On évite ainsi que l'utilisateur puisse confondre les commandes applicables aux dialogues structurés et celles qui sont utilisables pour la manipulation directe.

Enfin, outre les réactions de chaque type d'objet, l'on définit les services généraux que doit connaître l'interacteur de manipulation directe pour pouvoir fonctionner. Il s'agit des services

comme la fonction de désignation ainsi que les fonctions de sélection et dé-selection. Enfin, le concept de « poignée » est défini, afin de permettre d'associer des actions à des manipulations d'attributs de l'objet.

La nature déclarative de la description permet là encore d'obtenir des descriptions extrêmement concises, fort éloignées de la description sous forme d'automates que l'on rencontre souvent en manipulation directe.

- **Le modèle de présentation**

La présentation d'une AGI-CT peut se résumer à un (ou plusieurs) espace de visualisation du modèle, dont la gestion incombe principalement au noyau fonctionnel, et un ensemble de dispositifs permettant de définir les commandes de l'application. À ces dispositifs, on doit ajouter les widgets de base permettant la saisie de valeurs telles que des numériques ou des chaînes de caractères, et les dispositifs d'informations (fenêtre d'aide, écho de l'état du dialogue, etc.). Le modèle de présentation que nous avons utilisé est basé sur les tâches, et plus particulièrement sur la définition des moyens logiques permettant de déclencher des actions. De fait, le modèle de présentation définit des menus, sous-menus et des commandes. Ces dernières sont associées à une tâche, qu'elles permettent de déclencher. L'activation d'une commande de menu (utilisable par l'utilisateur) sera autorisée ou non par le contrôleur de dialogue en fonction des états respectifs des différents interacteurs.

Le modèle de la présentation se ramène donc à la description de menus, sous-menus et cases de commandes. Un menu est composé d'un label, qui apparaît dans la barre de menu ou dans le menu supérieur, et d'un ensemble de menus et boutons. Les boutons sont constitués soit d'un label, soit d'une image, et sont associés chacun à une tâche, définie dans le modèle de tâche.

3.2.4.2 GIPSE et couche graphique

Comme nous l'avons indiqué en introduction, le système GIPSE est capable de gérer une application à partir des fichiers de description des modèles présentés ci-dessus. Pour ce faire, il s'appuie sur une couche graphique spécialisée qui permet également au noyau fonctionnel de se visualiser. Le problème de l'interaction avec les objets du modèle ayant une représentation graphique est identifié par Myers [Myers 1995] comme un problème important pour les outils d'ingénierie de conception d'interfaces utilisateurs. Dans GIPSE, ce problème est résolu par l'application des principes suivants :

- La couche graphique fournit un espace de visualisation pour le modèle. Les pointés graphiques résultant des clics souris dans la surface visualisée sont récupérés par le contrôleur de dialogue dans l'espace du modèle, et le modèle est chargé de la désignation, qui correspond à la conversion de ces pointés en entités du modèle.
- La couche graphique fournit une notion de « commande » activable par l'utilisateur, au moyen des deux éléments « menu » et « bouton » du modèle de présentation de GIPSE. L'aspect externe de la présentation, et en particulier la disposition des menus et boutons sur l'écran sont ignorés de GIPSE. Ils peuvent être adaptés totalement indépendamment de l'application gérée par GIPSE.
- La disponibilité des commandes est automatiquement ajustée (commandes grisées) par le contrôleur de dialogue en fonction de l'état du dialogue. Le contrôleur de dialogue utilise là quelques fonctions spécifiques à la couche graphique

Moyennant ces différents points, il est possible d'utiliser GIPSE pour construire des applications graphiques interactives spécialisées. Ainsi, depuis plusieurs années, GIPSE sert-il d'outil de base aux projets de CAO des élèves de troisième année de l'ENSMA.

3.2.4.3 GIPSE : véritable système CAO pour applications interactives

Il peut sembler paradoxal de décrire un système interactif graphique par une procédure purement non interactive et textuelle, telle que la modification des fichiers texte représentant les modèles décrits dans la section précédente. En fait, GIPSE peut faire bien mieux.

Grâce à l'introduction des mécanismes décrits en 2.2.4.3, et à un contrôleur de dialogue entièrement dynamique, il est possible de construire avec GIPSE une véritable application graphique interactive par extension/spécialisation. Les modèles de GIPSE sont modifiables dynamiquement, ce qui permet en cours d'utilisation de modifier des interacteurs, de supprimer des commandes, d'ajouter des actions et des nouvelles commandes invoquant ces nouvelles actions, le tout dans une parfaite harmonie avec les actions « codées en dur » du système initial.

Les seules limites à l'extension du système concernent la possibilité de créer de nouvelles classes d'objets dynamiquement, ce qui fait l'objet d'un travail en cours au laboratoire [Texier & Guittet 1999].

3.2.5 En résumé

À partir d'un modèle d'architecture d'applications interactives appliqué au domaine de la CAO développé par L. Guittet, nous avons défini et étendu une stratégie de dialogue, les dialogues structurés, puis avons conçu un environnement de développement d'applications interactives, GIPSE.

3.2.5.1 Pour une plus grande utilisation des dialogues structurés

Le système de gestion du dialogue sous-jacent aux systèmes Like et EBP, appelé NODAO, mettait en œuvre le modèle H^4 et les interacteurs hiérarchisés [Girard, Pierra, & Guittet 1995]. Il générait le contrôleur de dialogue, à partir de descriptions externes sous forme d'ATN [Green 1986]. L'utilisation de ces systèmes a démontré tout l'intérêt des dialogues structurés, mais a également mis en évidence quelques-unes de leurs lacunes, tant vis-à-vis de l'utilisateur (complexité du dialogue, lourdeur pour certaines tâches) que du concepteur d'applications (verbosité des descriptions du dialogue, lourdeur du cycle de développement).

Pour résoudre les premiers problèmes, nous avons défini des solutions tant conceptuelles que techniques permettant d'obtenir un écho sémantique systématique multi-niveau et une intégration harmonieuse de deux formes de dialogue très différentes.

Pour faciliter la tâche du concepteur, nous avons défini des modèles déclaratifs beaucoup plus concis, et intégré la définition du dialogue de l'application et sa gestion pour obtenir un outil interactif de conception assistée d'applications graphiques interactives utilisant les dialogues structurés.

3.2.5.2 Publications référencées

Les travaux présentés dans ce chapitre ont été publiés dans une quinzaine de publications :

Les travaux sur la notion de dialogues structurés ont été diffusés dans [Girard, Pierra, & Guittet 1995 ; Guittet 1995 ; Guittet & Pierra 1993 ; Pierra, Girard, & Guittet 1995]. L'intégration de l'exploration a fait l'objet d'une démonstration [Patry & Girard 1998] et de plusieurs développements [Patry 1999b ; Patry & Girard 1997b].

L'ensemble des travaux sur les outils d'ingénierie des systèmes interactifs a donné lieu à [Chatty, Girard, & Sire 1996 ; Girard & Potier 1995 ; Patry 1996a ; Patry 1996b ; Patry 1998 ; Patry 1999a ; Patry & Girard 1997a ; Patry & Girard 1999].

3.3 Perspectives

Les perspectives ouvertes par notre travail peuvent se décliner selon deux axes : la validation et la diffusion des résultats obtenus d'une part, et l'enrichissement de l'environnement interactif de conception d'applications interactives d'autre part.

3.3.1 Validation et diffusion des dialogues structurés

Dans l'état actuel de nos travaux, il convient de valider les solutions que nous avons proposées. Malgré leurs avantages dans le domaine de la conception technique, il faut admettre que l'utilisation des dialogues structurés dans les systèmes CAO commerciaux demeure limitée. De nouvelles techniques, dites d'exploration, ont fait leur apparition, qui permettent de couvrir certains des aspects des dialogues structurés. Une étude exhaustive de ces mécanismes ainsi que du dialogue des systèmes CAO du marché permettrait de comparer les différentes approches. L'étude de l'applicabilité des dialogues structurés à la modélisation 3D par exemple, mais également à la navigation dans les mondes virtuels paraît intéressante.

Au demeurant, la diffusion des dialogues structurés ne se limite pas au domaine de la conception technique. En effet, il convient d'étudier les tâches auxquelles ces dialogues peuvent apporter un surcroît d'expressivité. La corrélation avec les études sur la visualisation des données complexes, et donc plus précisément sur l'interaction avec les données complexes, comme les données géographiques par exemple, semble naturelle. D'autres aspects, comme la modélisation géométrique ou la construction de scènes, peuvent être envisagés.

Sur le plan de l'ergonomie des dialogues structurés, il convient également de mener des études sur l'évaluation des solutions proposées, tant pour l'écho sémantique que pour l'intégration de plusieurs formes de dialogue. Peut-on mesurer un gain par rapport à l'utilisation de systèmes ne disposant pas de cette technique de dialogue ? L'utilisation de plusieurs formes de dialogue conjuguées engendre-t-elle plus d'erreurs ? Les utilisateurs délaissent-ils une forme au profit d'une autre ?

3.3.2 Enrichissement de GIPSE

GIPSE constitue un noyau de développement aux possibilités encore limitées. Sa diffusion et son extension sont deux points sur lesquels nous souhaitons plus particulièrement insister.

- **Plus large utilisation de la PsE / PbD**

La majorité des « Model-Based Systems » ont un cycle de développement non interactif. En effet, ils génèrent un système qui sera, lui, utilisé interactivement. Bien qu'utilisant des « fichiers texte » pour définir ses modèles, GIPSE est à même de modifier ces mêmes modèles en cours d'utilisation. Il est proche des principes gouvernant XXL [Lecolinet 1996] ; cependant, XXL nécessite une phase de recompilation en C pour effectuer certaines modifications concernant le comportement de l'interface.

GIPSE est capable de modifier ses différents modèles à partir de la création de nouvelles actions sur exemple, et de les réinterpréter. Mais ces modifications sont pour la plupart très stéréotypées. Pouvoir accéder à ces modèles pour les modifier de façon plus importante fournirait une interaction plus riche. L'adjonction d'outils de vérification (cf. section 4.2.2) permettrait également d'augmenter l'utilisabilité de ce système.

- **Modèle de tâche et interface**

Les travaux actuels sur les modèles de tâche sont nombreux. Des outils permettant de créer de tels modèles, mais également d'en étudier les possibilités, ont vu le jour ([Ballardin, Mancini, & Paternò 1999 ; Biere, Bomsdorf, & Szwillus 1999 ; Gamboa & Scapin 1997]). Cependant, ces outils concernent la définition et la simulation du comportement des modèles de tâche. Ils ne sont en aucune façon liés à l'application finale résultant de la réalisation. Alors qu'ils proposent le plus souvent une distinction entre tâches de l'utilisateur (abstraites) et tâches du système (concrètes), ils ne permettent pas de lier le modèle à l'application finalement conçue.

À l'inverse, les travaux sur l'évaluation ([Al-Qaimari & McRostie 1999 ; Farenc & Palanque 1999]) nécessitent de retrouver dans les actions de l'utilisateur ou dans les composantes de l'interface

les tâches effectives. L'absence de liens entre modélisation initiale et réalisation oblige à reconstruire les tâches à partir des interactions.

Il semble qu'il soit possible d'établir un lien entre modèle de tâche et réalisation, tant pour des besoins de prototypage que pour des études d'utilisabilité, aussi bien préalables qu'*a posteriori*. Ceci constituera d'ailleurs un lien avec certain travaux présentés au Chapitre 4.

- **Vers un multi-contrôleur de dialogue**

L'intégration de plusieurs modes de dialogue dans GIPSE ouvre la voie vers une extension des possibilités de dialogue des applications interactives. Les études préliminaires que nous avons menées, tant sur les applications collecticielles [Chatty, Girard, & Sire 1996] que sur la notion d'écho sémantique, font apparaître la nécessité de disposer d'un contrôleur de dialogue plus élaboré. La simplification des modèles déclaratifs utilisés doit cependant être recherchée, afin d'accroître l'utilisabilité des systèmes. Enfin, une totale bijection entre formalisme et outils graphiques doit être atteinte.

Ces travaux s'intègrent dans un projet plus vaste, visant à définir un générateur polyvalent d'applications interactives, qui sera plus détaillé dans notre projet de recherche (Chapitre 5).

Chapitre 4
Méthodes formelles pour les systèmes interactifs

Même si nous avons dès le début de nos travaux utilisé des notations formelles pour décrire de façon abstraite les dialogues de nos applications, ce thème de recherche est celui auquel nous nous sommes intéressé le plus récemment, soit en fait depuis 1996/97. Dans le cadre de cette synthèse, nous nous limiterons donc à un survol de l'utilisation des méthodes formelles dans les IHM, en insistant plus particulièrement sur les outils actuellement disponibles 4.1, puis en décrivant les travaux que nous avons engagés sur le domaine 4.2. Enfin, comme dans les autres chapitres, la dernière section donnera quelques perspectives 4.3.

4.1 Le domaine de recherche

Le domaine des méthodes formelles pour les systèmes interactifs est en pleine ébullition. Pour s'en convaincre, il n'est que de recenser l'ensemble des sessions et ateliers (« wokshops ») des principales conférences en IHM consacrées à ce sujet (ACM CHI, IFIP Interact, IFIP EHCI, BCS HCI, AFIHM IHM), mais également les conférences complètement dédiées à ce thème, comme FAHCI (« Formal Aspects of Human Computer Interface ») ou DSV-IS (« Eurographics Worshop on Design Specification and Verification of Interactive Systems »). Devant un tel nombre de travaux, il est illusoire de prétendre à l'exhaustivité, surtout si, comme c'est notre cas, les méthodes formelles dans les IHM ne constituent pas notre point focal d'intérêt.

Pour effectuer un survol de ce domaine, plusieurs approches sont possibles. La première consiste à partir d'une classification des notations et méthodes, comme par exemple celle établie dans [Brun & Beaudouin-Lafon 1995] qui utilise leur origine comme critère principal, donnant un véritable arbre taxinomique des formalismes pour les IHM. Une autre approche consiste, comme dans [Palanque 1997], à situer les différentes approches par rapport aux grandes phases du génie logiciel, pour séparer par exemple les méthodes traitant des besoins et de la prise en compte de l'utilisateur, de celles traitant de la spécification des systèmes ou de leur validation.

Il nous semble plus pertinent dans notre cas de choisir une présentation d'ordre plutôt chronologique. En effet, on peut différencier trois époques dans l'évolution des méthodes formelles pour les systèmes interactifs. La première, s'étendant jusqu'au début des années 1990, a vu un nombre important de créations de notations, dont la sémantique a été plus ou moins précisément définie (4.1.1). La seconde, qui recouvre partiellement la première puisqu'elle correspond en fait aux années 1990/97, a vu l'utilisation ou l'adaptation de méthodes formelles bien définies pour les appliquer à des problèmes particuliers dédiés aux IHM (4.1.2). Depuis l'article de Fields [Fields, Merriam, & Dearden 1997], on a pu relever une évolution des travaux vers une plus grande utilisabilité des méthodes formelles. L'utilisation combinée de méthodes, et surtout l'intégration ou le développement d'outils, marquent la période actuelle (4.1.3).

4.1.1 Notations et formalismes

Selon la taxinomie de [Brun & Beaudouin-Lafon 1995], on peut voir deux origines distinctes à l'utilisation des formalismes dans les IHM : la première est plutôt issue des informaticiens et se base sur la théorie du calcul (4.1.1.1), alors que la seconde est davantage centrée sur l'aspect utilisateur et provient des sciences cognitives (4.1.1.2).

Notons dès à présent un point important de terminologie. Nous appuyant sur la définition des spécifications formelles donnée dans [Gaudel et al. 1996], nous employons au cours de cette section les termes de *notation*, *spécification formelle* et *formalisme* :

« Une spécification est dite *formelle* si :
- elle est décrite en suivant une *syntaxe* bien définie, comme celle d'un langage de programmation ;
- la syntaxe est accompagnée d'une *sémantique* rigoureuse qui définit des modèles mathématiques représentant les réalisations acceptables de chaque spécification syntaxiquement correcte. »

Si l'ensemble des travaux présentés dans cette section dispose d'une syntaxe précise, le cas de la sémantique est beaucoup plus variable. Ceux décrits en 4.1.1.2 ne décrivent pas de modèle mathématique permettant de définir une sémantique rigoureuse. Nous les qualifierons du terme de *notation*. En revanche, de nombreux travaux répondent parfaitement à la définition ci-dessus, et seront qualifiés de *spécifications formelles* ou plus simplement *formalismes*.

4.1.1.1 Formalismes issus de la théorie du calcul

Nous suivons ici la présentation de [Brun & Beaudouin-Lafon 1995]. Les formalismes issus de ce domaine concernent plus spécialement les informaticiens. Il visent essentiellement la spécification du dialogue des applications interactives. On y trouve classiquement les grammaires hors contexte, les automates et autres réseaux de transition, et les modèles à événements. L'article célèbre de Green [Green 1986], mais aussi le livre de Olsen [Olsen 1992] permettent d'en faire une bonne synthèse.

Comme l'écrit Olsen [Olsen 1992], l'utilisation des grammaires hors contexte semble naturelle dans le cadre de l'interaction homme-machine, si l'on considère que traiter des événements d'entrée peut s'analyser comme du traitement de langage. Les menus et boutons remplacent les traditionnels mots-clefs et délimiteurs. [Hanau & Lenorovitz 1980], mais aussi Syngraph [Olsen 1983] ont ainsi exploité cette idée. Le dialogue y est défini par des règles de production ; les terminaux représentent les dispositifs logiques d'entrée (menus, boutons, etc.), alors que les non-terminaux définissent la structure des commandes.

Les automates ont probablement été les plus utilisés pour spécifier le dialogue des applications interactives (voir dans [Olsen 1992]), en raison en particulier de leur formalisme graphique. Les nœuds correspondent aux états de l'interface, alors que les transitions définissent l'enchaînement des actions. Leur principal défaut provient de l'explosion combinatoire des états lorsque croît l'application. Des variantes ont été définies, les RTN (« Recursive Transition Network ») ou les ATN (« Augmented Transition Network ») qui permettent d'accroître leur pouvoir d'expression tout en réduisant la taille de la description globale [Green 1986]. Un formalisme dérivé « orienté-objet », les « Statechart » [Harel 1988] a même été défini à partir des automates.

Issus comme les automates de la théorie des graphes, les réseaux de Petri permettent l'introduction d'aspects temporels. Ils présentent cependant les mêmes inconvénients que les automates sur le plan de l'utilisabilité pour de grandes applications. Palanque [Palanque 1992] a proposé une combinaison de la notion d'objet et des réseaux de Petri, qui a permis de résoudre une partie de ces problèmes.

Plus proches des grammaires, les systèmes à événements ont connu un grand succès car ils composent le cœur des couches graphiques et boîtes à outils actuelles. La définition de véritables formalismes à événements est cependant assez rare, mais on peut trouver certains exemples comme ERL (Event-Response Language [Hill 1986 ; Hill 1987]). En fait, une description en ERL est formellement équivalente à un automate à états finis [Olsen 1992]. Seule diffère la manière de spécifier le dialogue (par les événements plutôt que par les états).

Les formalismes présentés ci-dessus se sont dans leur ensemble uniquement intéressés, du moins dans un premier temps, à l'aspect contrôle du dialogue des applications interactives. Ce n'est qu'à partir du début des années 1990 qu'ils ont été étendus ou simplement utilisés pour décrire

d'autres aspects de l'interaction homme-machine. Des formalismes comme TAG [Schiele & Green 1990] et ETAG [de Haan 1995] ou encore les travaux de [Palanque & Bastide 1995] essaient ainsi d'appliquer aux tâches les méthodes issues de la théorie du calcul. Ces travaux constituent en quelque sorte la transition avec les modèles présentés en 4.1.1.2.

4.1.1.2 Notations en provenance des sciences cognitives

Les premiers travaux dans le domaine des formalismes décrivant l'activité de l'utilisateur sont certainement à mettre au crédit de Card, Moran et Newell [Card, Moran, & Newell 1983] et de leur modèle du processeur humain. CLG (Command Language grammar) [Moran 1981] est un modèle informel utilisant une notation symbolique. GOMS (Goals, Operators, Methods and Selection) [Card, Moran, & Newell 1983] est un modèle d'analyse de la tâche de l'utilisateur. Il a donné lieu à toute une famille de modèles associés [John & Kieras 1996], dont Keystroke [Card, Moran, & Newell 1983] par exemple, qui demeure encore aujourd'hui une référence pour l'analyse de l'interaction au plus bas-niveau.

Allant plus avant dans l'intégration des règles ergonomiques dans le processus de conception, MAD [Scapin & Pierret-Golbreich 1989 ; Scapin & Pierret-Golbreich 1990] offre un support pour le parallélisme et la synchronisation des tâches, que l'on peut mettre en parallèle avec MUSE [Lim & Long 1994], DIANE [Barthet 1988] ou encore HTA [Shepherd 1989].

À l'inverse des notations précédentes, UAN (User Action Notation) [Hartson & Gray 1992 ; Hix & Hartson 1993] s'attache à décrire les tâches à partir des interactions de bas-niveau, en séparant les actions de l'utilisateur, l'écho du système, et l'état de ce même système. Notons que UAN est défini par ses auteurs comme « extensible » et ne possède donc pas de sémantique propre.

Bien que non complètement formalisées, ces notations sont aujourd'hui très utilisées, et contribuent largement à combler le fossé (« bridging the gap ») existant entre le monde informel de l'utilisateur et l'espace formel de la programmation. Elles ont donné lieu à de nombreuses extensions, comme MAD* [Gamboa & Scapin 1997], XUAN [Gray, England, & McGowan 1994] ou encore DIANE^{+}[Tarby 1993].

4.1.2 Modèles formels

Les travaux présentés jusqu'ici s'intéressaient principalement à développer des formalismes adaptés au domaine des IHM, que ce soit pour exprimer le dialogue des applications ou pour construire un modèle des tâches de l'utilisateur. Ceux correspondant à ce que nous avons identifié comme la deuxième étape du développement des méthodes formelles dans les IHM ont abordé le problème par l'angle des propriétés qui pouvaient être démontrées ou vérifiées à l'aide des formalismes utilisés. Ceci a conduit à l'utilisation de formalismes pour atteindre de nouveaux objectifs ou à l'extension de formalismes existants. Mais il est symptomatique de remarquer que la majorité des travaux relevant de cette catégorie est à classer dans le troisième groupe de formalismes décrits dans [Brun & Beaudouin-Lafon 1995], à savoir ceux issus de la Théorie des Catégories. On peut dire que, par opposition à l'époque antérieure où le côté syntaxe a été privilégié, c'est ici le côté sémantique qui est mis en avant.

Un des premiers résultats de ces travaux a été la définition formelle de la notion d'interacteur, dans l'objectif de construire des systèmes sur des briques de base formellement définies. L'utilisation de Z [Spivey 1988] par l'équipe de York [Duke & Harrison 1993] et celle de LOTOS en Italie [Paternò 1994 ; Paternò & Faconti 1994] ont ainsi permis d'établir les bases de cette notion. Les travaux de [Palanque & Bastide 1994] définissant le modèle des ICO ont donné une autre définition formelle à base de réseaux de Petri. Puis, la mise en commun des résultats obtenus a permis d'obtenir une définition plus universelle [Duke, et al. 1994], voire une méthode unifiée de conception à base d'interacteurs [Palanque, et al. 1996]. Même si des modèles d'interacteurs sont encore définis aujourd'hui [Accott, et al. 1997 ; Faconti & Duke 1996], aucune boîte à outils n'a, à ce jour, été écrite complètement en utilisant l'un de ces formalismes.

De fait, en dehors des aspects liés aux interacteurs, l'idée générale de preuve par construction de programme a été peu utilisée dans les IHM . Tout au plus peut-on relever les travaux avec HOL [Bumbulis, et al. 1996] et Object-Z [Hussey & Carrington 1997], mais qui demeurent encore très partiels.

La vérification des systèmes interactifs à l'aide de techniques de « model-checking » a été beaucoup plus développée [Campos & Harrison 1997]. Une première approche [Abowd, Wang, & Monk 1995] utilise CTL et l'outil SMV (Symbolic Verifier Tool), alors que [Paternò & Mezzanotte 1995] utilisent ACTL à partir de descriptions d'interacteurs faites en LOTOS, et [d'Ausbourg 1998 ; d'Ausbourg, Durrieu, & Roché 1996] utilisent LUSTRE.

Enfin, citons quelques langages définis spécialement pour les systèmes interactifs, comme Gralpla [Torres, et al. 1996] dans le domaine des spécifications algébriques, ou XTL [Brun 1997] dans celui de la logique temporelle.

4.1.3 Vers une véritable utilisation d'outils formels

L'étape logique, après une étude de faisabilité sur des cas d'école ou très limités, consiste à essayer d'appliquer les méthodes étudiées sur des cas réels. Pour ce faire, l'utilisation d'outils est nécessaire. Beaucoup de travaux sont en cours aujourd'hui pour créer une méthodologie de développement formel avec des outils existants, ou pour créer des outils permettant d'utiliser les méthodes définies à la section précédente. Une dernière voie consiste à créer des outils permettant de passer automatiquement ou semi-automatiquement d'un formalisme à l'autre pour conduire des analyses.

IOG (Interaction Object Graphs) [Carr 1997], qui définit graphiquement les dialogues sous forme d'automates étendus, offre un premier pas en fournissant une bibliothèque d'objets directement « linkables » avec l'application. [Hussey & Carrington 1998] définissent un ensemble de modèles pour passer de spécifications formelles en Object-Z à des *widgets* classiques. Les auteurs envisagent de créer un outil pour aider l'opération.

[Campos & Harrison 1999] ont entrepris de développer un outil semi-automatique pour transformer des interacteurs spécifiés en MAL (Modal Action Logic) [Ryan, Fiadeiro, & Maibaum 1991] en SMV pour effectuer des vérifications en « model-checking ». Un « theorem prover », PVS est également utilisé pour prouver des propriétés sur la présentation exprimées sous forme de spécifications algébriques. Notons que cette approche consiste à travailler formellement sur des modèles de l'application.

À l'inverse, d'Ausbourg [d'Ausbourg 1998 ; d'Ausbourg & Cazin 1999] travaille sur le système réel, à partir duquel il extrait une spécification LUSTRE qu'il combine à une approche en logique temporelle (TRIO) pour générer des tests du système.

Dans une approche plus constructive, [Cabrera, Torres, & Gea 1999] développent un outil de prototypage permettant de générer à partir de spécifications algébriques, d'une part la partie non interactive du code qui est compilée directement (C^{++}), et d'autre part un code Java ou Tcl/Tk que l'utilisateur peut ensuite reprendre avec un constructeur d'interface pour en modifier la présentation.

Enfin, nous ne saurions oublier Paternò, qui développe, dans le cadre d'un projet européen, le projet GUITARE, une série d'outils, CTTE (« Concur Task Tree Environment ») [Ballardin, Mancini, & Paternò 1999] autour du formalisme de description de modèles de tâches ConcurTaskTree, et intégrant les travaux effectués antérieurement sur LOTOS et ACTL (http://giove.cnuce.cnr.it/ctte.html) et Palanque, qui développe dans le cadre du projet européen MEPHISTO une série d'outils permettant d'utiliser le formalisme des ICO.

4.2 Contribution

Comme nous l'avons déjà mentionné, les travaux concernant ce domaine en sont encore à un stade exploratoire. Nous ne cherchons pas à créer un nouveau formalisme, mais plutôt à utiliser une

méthode formelle sur des cas concrets, comme nous l'avons fait pour la PsE/PbD dans le domaine de la CAO. Pour ce faire, les trois voies décrites dans la section 4.1.3 ont été abordées. Dans un premier temps, nous avons participé à la définition du modèle des interacteurs de dialogues, et formalisé ce modèle avec EXPRESS (4.2.1). La deuxième étude a consisté à valider un système réel en partant des tâches de l'utilisateur (4.2.2). Enfin, nous avons appliqué une technique de développement formel par construction (la méthode B) à un système de manipulation directe, pour lequel nous avons réalisé toutes les étapes depuis les spécifications jusqu'à l'exécution du code (4.2.3). À l'inverse des autres chapitres, compte tenu de leur nature plutôt exploratoire, nous ne proposerons pas un récapitulatif des travaux effectués et publiés, mais indiquerons en fin de chaque section, la diffusion des résultats obtenus.

4.2.1 Formalisation EXPRESS des Interacteurs hiérarchisés

Le modèle des interacteurs hiérarchisés a été présenté par L. Guittet [Girard, Pierra, & Guittet 1995 ; Guittet 1995]. Dans ce modèle, qui constitue une implémentation du modèle H^4, le contrôleur de dialogue dispose d'une hiérarchie d'agents réactifs, nommés *Interacteurs de contrôle*, lesquels correspondent aux différents niveaux de tâches et sous-tâches que peut accomplir le système. Chaque interacteur de contrôle est associé à des types d'informations d'entrée définis par les tâches qu'il permet d'accomplir. Il produit des informations de sortie qui représentent le résultat de ces tâches. Les informations de sortie d'un interacteur de contrôle peuvent constituer les informations d'entrée pour un autre interacteur. Une hiérarchie d'interacteurs permet ainsi de gérer un dialogue structuré tel que nous l'avons défini dans le chapitre 3. Cette notion d'interacteur s'intègre dans l'ensemble des travaux présentés en section 4.1.2. On notera cependant une différence importante : alors que les interacteurs de Duke et Paternò sont très proches des *widgets* et intègrent des éléments de présentation, les interacteurs de contrôle que nous avons définis demeurent au niveau du contrôle de dialogue et n'utilisent donc que des jetons abstraits. Ils sont en fait plus proches des *transducers* de [Accott et al. 1997], mais s'en distinguent par la définition des relations qu'ils ont avec le modèle de l'application.

Nous avons formalisé la notion d'interacteur de contrôle à l'aide d'EXPRESS [Bouazza 1995b ; EXPRESS 1994], langage de spécification formelle de modèles de données. Défini dans le cadre du projet STEP (Standard for the Exchange of Products) [Bouazza 1995a], EXPRESS est un langage orienté-objet qui bénéficie de nombreux outils, qui permettent par exemple de le compiler.

4.2.1.1 Description succincte d'EXPRESS

EXPRESS ne modélise que les aspects statiques des données. Il utilise à cet effet un ensemble d'entités structurées par des relations provenant du modèle objet (héritage, liens, cardinalités, etc.) et possédant des attributs. Un des points les plus importants du langage est la possibilité de définir des contraintes sur les attributs en logique du premier ordre.

Ces contraintes sont locales (appliquées à chacune des instances d'une entité) ou globales (applicables à toutes les instances d'une ou plusieurs entités). À la différence de la plupart des systèmes de modélisation de données comme le modèle entité-association [Chen 1976] ou OMT [Rumbaugh, et al. 1991] qui ne s'intéressent qu'aux cardinalités et à quelques contraintes inhérentes au modèle (notion de clé par exemple), EXPRESS permet de créer toutes sortes de contraintes. De plus, des fonctions de dérivation, écrites dans un langage ressemblant à PASCAL, peuvent être définies.

Outre la possibilité de décrire un modèle de données, comme nous l'avons exposé ci-dessus, EXPRESS permet de décrire un ensemble d'instances des entités modélisées, constituant ce que l'on appelle les « fichiers physiques ». Les outils disponibles, comme ECCO [Staub & Maier 1992] que nous avons utilisé, permettent de vérifier la conformité des fichiers physiques par rapport à une modélisation donnée.

Nous décrivons ci-dessous quelques éléments du langage directement à partir de la formalisation des interacteurs.

4.2.1.2 Formalisation des interacteurs en EXPRESS

La formalisation se fait en quatre étapes. Les jetons du dialogue doivent d'abord être définis (tokens), puis le système lui-même. Ensuite, nous représentons les interacteurs, et leurs composants élémentaires.

- **Les jetons ou tokens**

Comme nous l'avons vu dans le chapitre 2, les jetons peuvent être modélisés par un arbre d'héritage. En fait, ils peuvent être à la base de deux types, commande ou paramètre :

```
ENTITY token ;
  ABSTRACT SUPERTYPE OF ( ONEOF ( command, parameter ) ) ;
END_ENTITY ;
```

Alors que l'entité *command* contient un attribut constitué d'une chaîne de caractères dont la valeur doit être unique dans le modèle, l'entité *parameter* comme *token* est une entité abstraite. On peut ainsi définir des arbres d'héritage. On notera l'apparente redondance dans la définition (*supertype of / subtype of*). La déclaration SUPERTYPE permet de définir si les sous-types sont mutuellement exclusifs (ONEOF) ou non (AND et ANDOR).

```
ENTITY command
  SUBTYPE OF (token) ;
  name : STRING ;
  UNIQUE name;
END_ENTITY ;

ENTITY parameter;
  SUBTYPE OF (token) ;
  ABSTRACT SUPERTYPE OF ( ONEOF ( number, position, object ) ) ;
END_ENTITY ;
```

La suite de la définition permet de compléter l'arbre d'héritage. On notera la référence à des éléments contenus dans le modèle du noyau fonctionnel (non décrit ici).

```
ENTITY number
  SUBTYPE OF (parameter) ;
  value : REAL ;
END_ENTITY ;

ENTITY position
  SUBTYPE OF (parameter) ;
  point : LIST [1:3] OF real ;
END_ENTITY ;

ENTITY object;
  SUBTYPE OF (parameter) ;
  SUPERTYPE OF ( ONE OF ( line, circle, ... ) ) ;
  reference : BINARY ;
END_ENTITY ;
```

- **Le système global**

Le système global regroupe les différents éléments (interacteurs et jetons) et définit leurs relations. L'entité *system* est composée d'un ensemble de jetons (utilisés par les interacteurs : **SET**[0:?] **OF** token), d'un sous-ensemble de jetons produits par la couche de bas-niveau (*base_production*), et d'une liste ordonnée d'interacteurs (**LIST**[0:?] **OF** interactor). Cet ordre est défini par la logique de production/consommation des jetons.

```
ENTITY system;
  tokens: SET [0:?] OF token ;
  base_production: SET [0:?] OF token ;
  hierarchy: LIST [0:?] OF interactor ;
  WHERE
    Base : base_production IN tokens ;
    production : QUERY ( i <* hierarchy ½ i.token_in IN
        base_production + production_behind ( SELF, i ) ) = hierarchy ;
END_ENTITY ;
```

Les contraintes définies dans la clause WHERE expriment que chaque interacteur (QUERY()=hierarchy) doit être capable de recevoir les jetons produits par les interacteurs situés en dessous dans la hiérarchie. Cette règle utilise le résultat d'une (FUNCTION *production_behind*) qui

calcule l'ensemble des jetons susceptibles d'être produits par les interacteurs situés en dessous de celui passé en paramètre :

```
FUNCTION production_behind( s: system ; i: interactor ) : SET OF token ;
  LOCAL tokens: SET OF token := [] ;
        ni : INTEGER := 1;
  END_LOCAL;
  REPEAT UNTIL ( i = s.hierarchy[ni] ) ;
    tokens := tokens + s.hierarchy[ni].product ;
    ni := ni + 1;
  END_REPEAT;
  RETURN tokens ;
END_FUNCTION;
```

· **Les interacteurs**

Les interacteurs sont décrits par des réseaux de transition (ATN) :

```
ENTITY interactor  ;
  name : STRING ;
  states : SET [0:?] OF state ;
  transitions : SET [0:?] OF transition ;
  initial_state : state ;
  DERIVE
    token_in : QUERY ( t <* syst.tokens ½ SIZEOF (
                           QUERY ( tr <* transitions ½ tr.token = t ) # 0) ;
    Product  : QUERY ( t <* syst.tokens ½ SIZEOF (
                           QUERY ( tr <* transitions ½ tr.action.production = t) # 0) ;
  INVERSE syst : SET [1:1] OF system FOR hierarchy ;
  UNIQUE name ;
  WHERE initial_state IN states;
END_ENTITY ;
```

Chaque interacteur est donc composé d'un nom et des états et transitions correspondant à l'ATN qui décrit son comportement. Deux caractéristiques principales sont calculées par dérivation (DERIVE) : les jetons consommés par l'interacteur (*token_in*) et ceux qu'il produit (*product*).

· **États, transitions et actions**

Les états, transitions et actions complètent la définition des ATN

```
ENTITY state ;
  name : STRING ;
  prompt : OPTIONAL STRING ;
  INVERSE inter : LIST [1:1] OF interactor FOR states ;
  UNIQUE inter , name ;
END_ENTITY ;
```

Les états correspondent aux états stables de l'interaction : le système attend une entrée de l'utilisateur. L'attribut optionnel *prompt* est un message qui peut être envoyé à l'utilisateur pour le guider dans son interaction.

```
ENTITY action  ;
  in_list : LIST [0:?] OF parameter ;
  production : OPTIONAL parameter ;
END_ENTITY ;
```

Les actions définissent l'interface d'appel des fonctions du noyau fonctionnel. Cette interface est réduite aux paramètres d'entrée et de sortie.

```
ENTITY transition  ;
  s_begin, s_end : state ;
  key : token ;
  action : OPTIONAL action  ;
  INVERSE inter : SET [1:1] OF interactor FOR transitions ;
  WHERE
    inter_coherency : inter = s_begin.inter AND inter = s_end.inter ;
END_ENTITY ;
```

Les transitions permettent de connecter les états, et peuvent déclencher des actions (ATN). Nous détaillons ci-dessous les vérifications de cohérence qui peuvent être menées.

4.2.1.3 Utilisation du modèle EXPRESS

Notre but premier était d'utiliser la modélisation EXPRESS pour vérifier la correction du contrôleur de dialogue. Un contrôleur de dialogue particulier s'exprime sous la forme d'un fichier physique devant être confronté au modèle ci-dessus. La vérification à l'aide d'ECCO de ces fichiers

physiques permet donc de garantir les propriétés exprimées par le modèle. Cette section présente quelques exemples des propriétés qui peuvent être ainsi vérifiées, lesquelles sont en fait de deux types : elles peuvent être localisées à un seul interacteur, ou être globales et s'exprimer entre plusieurs interacteurs, à travers les échanges de jetons.

- **Exemples de propriétés locales aux interacteurs**

À un niveau structurel, les contraintes définies (clauses WHERE) permettent d'assurer la connectivité et l'atteignabilité du réseau de transition. Pour détailler le niveau logique, nous employons une métaphore langagière. Au niveau lexical, on peut vérifier que chaque commande ou paramètre défini dans la liste des éléments d'entrée de l'automate est effectivement utilisé par ce dernier. Au niveau syntaxique, l'absence d'ambiguïté du réseau peut être vérifiée (deux transitions possédant la même clé ne peuvent pas partir du même état). Au plan sémantique, on peut vérifier que la liste des paramètres d'entrée des actions est consistante avec les paramètres disponibles dans l'interacteur lors de leur appel.

- **Exemples de propriétés globales**

Des propriétés globales au système peuvent également être vérifiées. Par exemple, chaque jeton utilisé par un interacteur doit pouvoir être produit par au moins un des interacteurs situé en dessous de lui dans la hiérarchie (ou par la liste des jetons produits au bas-niveau). Cette règle a été décrite dans la description du système. De la même façon, on peut assurer que la règle inverse s'applique : tout jeton produit par un interacteur peut être consommé par un interacteur qui est situé au-dessus. Notons bien que toutes ces règles sont des règles statiques.

- **Utilisation de l'outil ECCO**

Nous avons utilisé l'outil ECCO pour valider cette approche. Nous avons construit un modèle de données complet représentant les interacteurs hiérarchisés, et nous avons écrit un fichier physique représentant une instance de petit système CAO. Toutes les propriétés énoncées ci-dessus ont pu être vérifiées.

Les travaux sur EXPRESS et son utilisation en CAO peuvent être retrouvés en particulier dans [Aït Ameur, et al. 1995 ; Pierra, et al. 1996]. Le travail présenté dans cette section a été publié dans [Guittet, Girard, & Pierra 1997].

4.2.2 La validation des systèmes interactifs

De nombreux auteurs cherchent à concevoir, puis à valider, les systèmes interactifs à partir de modèles de tâches [Gamboa & Scapin 1997 ; Paternò, Mancini, & Meniconi 1997 ; Tarby 1993]. Ceci est possible lorsque ce dernier peut être construit. Or, compte tenu de leur nature, les systèmes CAO, au même titre que tous les « éditeurs » [Fekete 1996b] ne permettent pas d'envisager la construction d'un tel modèle. Pourtant, la validation de ces systèmes serait bien utile dans certains cas. Citons deux exemples :

- Pour acquérir un système (en particulier aussi cher que les systèmes CAO actuels) on peut supposer qu'un acheteur aimerait être certain que ce qu'il veut réaliser peut effectivement se faire sur le système qu'il évalue. Une validation de scénarios serait intéressante.
- Lors d'une mise à disposition d'une nouvelle version, il est important de vérifier que ce que l'on savait faire avec l'ancienne version est toujours possible avec la nouvelle. Une validation automatique serait ici bienvenue.

Dans ces deux cas, il ne s'agit pas de demander une validation de l'ensemble des tâches possibles sur le système, mais de valider certaines d'entre elles, semi-automatiquement ou automatiquement. Le but de notre travail était de fournir une réponse à ce besoin dans le cadre des applications à dialogue structuré.

4.2.2.1 Description des tâches à tester

L'idée générale de la méthode que nous avons employée consiste à décrire un certain nombre de tâches susceptibles d'être réalisées par le système, puis de générer à partir de cette description des vecteurs de tests. Ces vecteurs de tests sont ensuite fournis à une version modifiée de l'application qui peut les « exécuter » et donc les valider.

Le langage de description de tâches que nous avons utilisé est une version simplifiée de MAD* [Hammouche 1995], limitée aux seuls opérateurs SEQ, OR et AND :

- SEQ signifie que chaque sous-tâche doit s'exécuter une fois, dans l'ordre défini par l'écriture,
- OR signifie que l'une des tâches doit s'exécuter,
- AND signifie que les tâches doivent s'exécuter dans un ordre quelconque.

Un exemple de tâche est donné par la description graphique de la Figure 7. Il s'agit de créer un cercle dont le centre est donné par la projection d'un point sur un segment, et dont le rayon peut être donné par un réel ou par une expression correspondant à la multiplication d'un réel par la distance entre un point et l'extrémité du même segment.

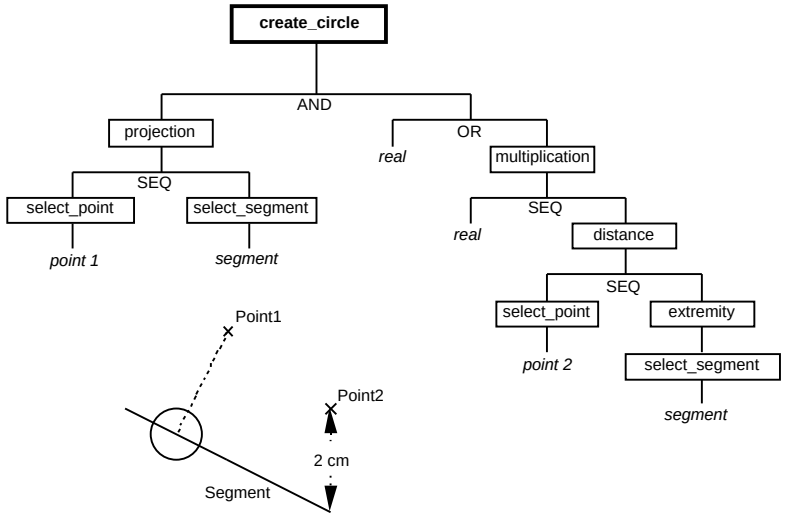

Figure 7 : Création d'un cercle

La traduction de cet arbre de tâches/sous-tâches donne le fichier suivant, où les mots-clés AND et OR sont respectivement remplacés par les symboles & et / (le mot-clef SEQ est remplacé implicitement par l'enchaînement des paramètres) :

```
## task 1

      command : create_circle
      result : circle
      parameter : position=<11>
      & parameter : real=a_real / <13>

      # sub-task 11
             command : projection
             result : position
             parameter : point=<111>
             parameter : segment=<112>
      # end sub-task

      # sub-task 13
             command : multiplication
             result : real
             parameter : real=a_real
             parameter : real=<12>
      # end sub-task

      # sub-task 12
             command : distance
             result : real
             parameter : point=<121>
             parameter : position=<122>
```

```
      # end sub-task

      # sub-task 122
             command : extremity
             result : position
             parameter : segment=<1221>
      # end sub-task

      # sub-task 111
             command : select_point
             result : point
             parameter: position=a_position
      # end sub-task

      # sub-task 112
             command : select_segment
             result : segment
             parameter : position=a_position
      # end sub-task

      # sub-task 121
             command : select_point
             result : point
             parameter : position=a_position
      # end sub-task

      # sub-task 1221
             command : select_segment
             result : segment
             parameter : position=a_position
      # end sub-task

## end task
```

À cette étape, nous avons développé un outil automatique qui génère toutes les séquences possibles d'interactions. Celles-ci sont en nombre fini, car nous n'avons pas introduit de répétition dans la notation. Seules des alternatives sont présentes, explicitement avec le mot-clef OR, et implicitement avec le mot-clef AND (on génère les deux séquences possibles). Dans la portion de fichier ci-dessous, deux des quatre séquences possibles à partir de la tâche décrite en Figure 7 sont décrites. Les séquences sont bornées par les mots SEQUENCE et END_SEQUENCE, et le mot-clef COMMAND est ajouté devant les noms de commandes du modèle de tâche.

```
SEQUENCE
COMMAND : create_circle
a_reel
COMMAND : projection
COMMAND : select_point
a_position
COMMAND : select_segment
a_position
END_SEQUENCE

SEQUENCE
COMMAND : create_circle
COMMAND : projection
COMMAND : select_point
a_position
COMMAND : select_segment
a_position
COMMAND : multiplication
a_real
COMMAND : distance
COMMAND : select_point
a_position
COMMAND : extremity
COMMAND : select_segment
a_position
END_SEQUENCE

…
```

Il est ainsi relativement aisé de définir un nombre important de vecteurs de tests.

4.2.2.2 Le test du système

Pour effectuer les tests, nous avons modifié légèrement le noyau d'une application utilisant GIPSE. Sans toucher au dialogue de l'application (c'est lui que nous voulons tester), nous avons fourni un deuxième mode d'entrée des données, en permettant la lecture des jetons d'entrée dans un fichier texte. Lorsque l'application est en mode test, elle ne demande plus ses données d'entrées par l'intermédiaire de la couche graphique, mais lit des données dans les fichiers fournis en paramètres. Les données d'entrée étant simplifiées (elles ne comprennent pas de valeurs), il convenait également de modifier l'appel au noyau fonctionnel. À ce niveau, l'appel ne s'exécute pas, mais une ligne est écrite dans un fichier de sortie pour effectuer la trace des actions réalisées.

Dans la version expérimentée, la corrélation entre commandes et paramètres du fichier d'entrée et jetons du système testé se fait au moyen d'un fichier de transcodage défini manuellement.

```
command : create_circle = COMMANDE : creation.cercle
command : create_segment = COMMANDE : creation.segment
command : create_point = COMANDE : creation.point
command : multiplication = COMMANDE : calc.multiplication
command : projection = COMMANDE : calculs.projection
command : distance = COMMANDE : calculs.distance
command : select_circle = COMMANDE : designation.cercle
command : select_segment = COMMANDE : designation.segment
command : select_point = COMMANDE : designation.point
command : extremity = COMMANDE : information.extremite
…
a_real : = REEL :
a_position : = POSITION :
a_segment : = SEGMENT :
a_point : = POINT :
a_circle : = CERCLE :
…
```

Les résultats sont également fournis sous forme de fichier texte. Le système est capable automatiquement de détecter un certain nombre d'erreurs, comme l'absence d'invocation d'une action au moment où l'analyse de tâche en prévoit une, ou encore des paramètres inconsistants (en ordre ou en nombre).

4.2.2.3 Résumé de l'étude

Cette étude concerne donc la validation du dialogue homme-machine d'une application interactive à partir d'une analyse de la tâche de l'utilisateur. Le processus de validation se base sur une génération automatique de vecteurs de tests à partir de l'analyse de la tâche. Ces vecteurs sont ensuite injectés à un simulateur qui reprend le code exécutable du Contrôleur de Dialogue de l'application à tester. Le simulateur vérifie ainsi l'atteignabilité et la complétude de l'interaction, et permet de valider le dialogue réel de l'application.

Cette étude montre qu'il est possible de vérifier automatiquement et directement sur le système final la conformité entre une analyse de la tâche d'un utilisateur et son implémentation dans le système final. Cette validation présente deux intérêts majeurs : tout d'abord elle s'effectue entre les deux extrémités du cycle de conception du logiciel, d'un côté le cahier des charges, et de l'autre le logiciel final. En outre, elle peut s'effectuer a posteriori sur un logiciel déjà opérationnel dont on ne modifie qu'une faible partie du code exécutable.

Cette étude a été menée dans le cadre du DEA de Y. Boisdron, et a donné lieu à une publication [Jambon, Girard, & Boisdron 1999].

4.2.3 Utilisation de la méthode B

L'objectif de l'étude que nous avons menée avec B était de tester l'utilisation de la méthode B [Abrial 1996] et de l'outil Atelier B [Steria Méditerranée 1997] sur un cas d'étude déterminé au sein du groupe de travail FLASHI du GDR-PRC Communication Homme-Machine. Le cas d'étude est un système de Post-It Notes®[6] coopératif, qui utilise en particulier pleinement la manipulation

6 Post-It Notes® est une appellation déposée par 3M

directe. Dans un premier temps, nous expliquons les motivations qui nous ont conduit à choisir la méthode B (4.2.3.1) ; Dans un deuxième temps, nous décrivons le cas d'étude (4.2.3.2). Puis, nous présentons l'utilisation de B sous l'angle de la méthode de développement (4.2.3.3) et sous celui des propriétés au sens IHM (4.2.3.4).

4.2.3.1 Motivations du choix de la méthode B

Le choix de la méthode B repose sur plusieurs critères. Il apparaît clairement dans la synthèse que nous avons faite en 4.1 que les méthodes formelles constructives utilisant un système de preuve ont été peu employées, en grande partie par manque d'outil. Le plus souvent, les vérifications de propriétés s'effectuent sur des modèles des applications, et rien ne garantit que les applications réelles qui en sont issues les respectent exactement. On peut ainsi relever les avantages suivants à l'utilisation de B :

- La méthode B autorise un développement complet, des spécifications les plus abstraites au code final. De plus, comme nous le verrons, ses principes la rapprochent des habitudes de programmation dans les IHM, et il est en particulier relativement aisé d'intégrer les boîtes à outils dans le processus de conception.
- L'outil Atelier B [Steria Méditerranée 1997] permet de s'assurer à chaque pas du développement que les obligations de preuves générées par le développement sont vérifiées, et ce à tous les niveaux de raffinement. Nous verrons en 4.2.3.3 comment la réalité du développement conduit à l'adapter.
- La nature de la méthode B fait qu'il n'y a pas de vérification de propriétés à proprement parler. Les propriétés du logiciel sont assurées par construction. Il est donc nécessaire de les prévoir pendant la phase de développement. De plus, la bonne modularité de B fait que les propriétés peuvent être correctement réparties selon les modules de l'application, suivant en cela les recommandations des modèles d'architecture propres au domaine.
- Le dernier argument, et non des moindres, tient en l'opérabilité de la méthode B, qui a démontré sur des cas réels non triviaux toutes ses possibilités.

4.2.3.2 Le cas d'étude

Le cas d'étude est un logiciel devant étendre les possibilités des Post-It Notes® pour prendre en compte des aspects collaboratifs. L'application se compose d'un bloc dont on peut détacher des feuillets (appelés « notes » par la suite), et d'icones représentant les destinataires des messages. Seules sont décrites les opérations de manipulation directe. Le bloc peut être iconifié, mais pas déplacé. Il présente trois zones actives en manipulation directe, pour l'iconifier, le fermer, et détacher une note. Les icones sont de trois types, pour représenter les trois objets de l'application.

Les notes ont le même aspect que le bloc, à l'exception de la zone de modification de taille. Leur comportement est différent de celui du bloc dans le fait que l'on peut les détacher du bloc et les déplacer (à l'aide de la zone qui sert à détacher une note sur le bloc), et les « déposer sur » une icone de destinataire pour les envoyer. On ne peut en revanche pas les remettre sur le bloc. Enfin, on peut entrer du texte aussi bien à l'intérieur des notes que directement sur le bloc.

Nous ne détaillons pas ici tous les aspects liés à la collaboration, notre étude ayant essentiellement porté sur les aspects non collaboratifs, même si l'architecture modulaire de l'application fait apparaître tous les modules (« machines » en B). La Figure 8 donne un aperçu de ce que pourrait être une implémentation de cette application. Les zones actives en manipulation directe sont montrées.

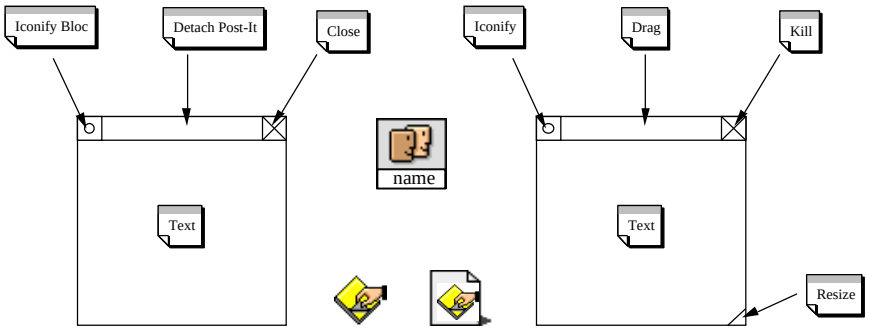

Figure 8 : (de gauche à droite) le bloc de Post-It Notes®, les trois types d'icones et la note elle-même.

4.2.3.3 Le développement en B

Le développement en B se base sur la notion de « machines abstraites » [Abrial 1996], et ce, de façon uniforme, que l'on soit au niveau le plus élevé (et abstrait) de la spécification ou au niveau du code. Seul un typage des machines, associé à la définition de structures de langage différentes, permet de différencier les étapes du raffinement. Ainsi, les mots-clefs sont-ils respectivement MACHINE, REFINEMENT et IMPLEMENTATION. Comme pour EXPRESS, la description d'un langage comme B sort du cadre de ce mémoire, c'est pourquoi nous nous contenterons d'expliquer le formalisme au fur et à mesure de son apparition dans l'exemple détaillé.

Afin de donner un cadre familier au lecteur du domaine des IHM, nous avons replacé les différents modules qui composent l'application dans le schéma général de ARCH [Bass, et al. 1992]. Sur la Figure 9, les ovales représentent les machines B, alors que les flèches représentent les relations de visibilité entre machines. Elles se traduisent en une clause du langage B appelée « EXTEND », qui rend visible à la machine qu'il utilise tous les éléments contenus dans l'autre machine. Compte tenu de la sémantique de B, les propriétés décrites sur la machine « étendue » sont conservées.

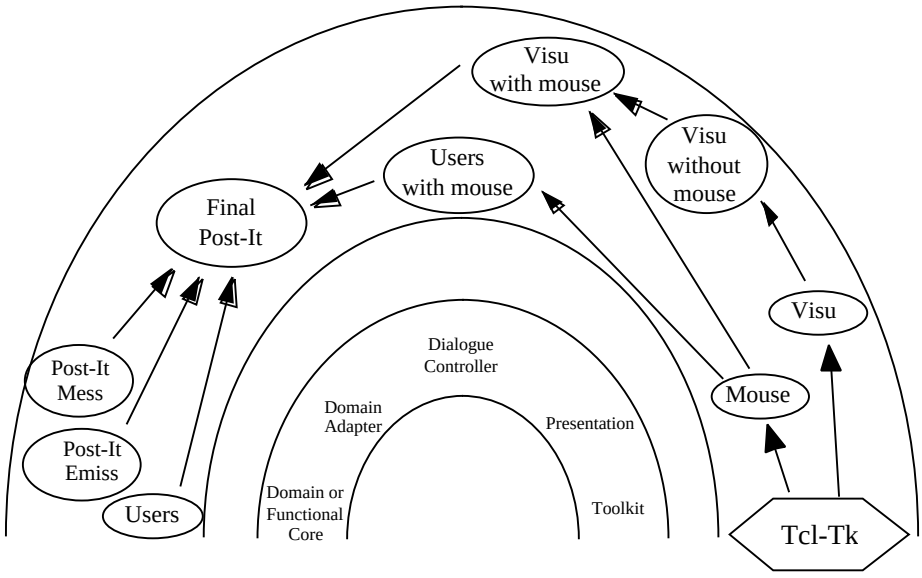

Figure 9 : Relations entre machines, présentées selon l'architecture ARCH

Les trois machines situées le plus à gauche gèrent le noyau fonctionnel de l'application, à savoir le contenu des messages (*Post-It_Mess*) et leur statut vis-à-vis de l'émission collecticielle (*Post-It_Emiss*), ainsi que les utilisateurs (*Users*). Les machines *Mouse* et *Visu* correspondent à la rétro-conception pour les besoins de l'application de la boîte à outil utilisée, en l'occurrence Tcl/Tk. Les deux machines *Visu_without_Mouse* et *Visu_with_Mouse* gèrent la visualisation des Post-It Notes® et leur interaction à l'aide de la souris. La gestion plus simple des utilisateurs a conduit à ne pas différencier deux machines dans leur cas. Enfin, la machine *Post_It_Final* assemble toutes les machines pour constituer l'application. Dans la suite de cette section, nous allons illustrer le

développement en B en donnant quelques exemples tirés des machines *Mouse* et *Visu_xx_Mouse*. Les extraits présentés proviennent de l'exemple traité, disponible en intégralité dans [Aït-Ameur, Girard, & Jambon 1998b]. Ils ont été simplement modifiés pour faire apparaître des symboles mathématiques plus « classiques » que ceux utilisés par l'Atelier B.

· **Rétro-conception de la souris**

La spécification de la machine gérant la souris est suffisamment simple pour être facilement compréhensible, mais permet néanmoins d'illustrer convenablement les propriétés qui peuvent être gérées par B. La première partie définit l'ensemble des souris possibles (`POST_MOUSE`), l'ensemble des états possibles pour les souris (`MOUSE_STATE`), le nombre maximal de souris possibles (`max_post_mouse`), les dimensions de l'écran (`max_mouse_pos_width, max_mouse_pos_high`) et les positions par défaut (`x_mouse_pos_default, y_mouse_pos_default`). Notons que nous avons modélisé ici uniquement un bouton.

```
MACHINE POSTIT_MOUSE
SETS
  POST_MOUSE;
  MOUSE_STATE = { up, down, clicked }
CONSTANTS
  max_post_mouse,
  x_mouse_pos_default, y_mouse_pos_default,
  max_mouse_pos_width, max_mouse_pos_high
PROPERTIES
  max_post_mouse = card(POST_MOUSE) &
  x_mouse_pos_default = 20 &
  y_mouse_pos_default = 20 &
  max_mouse_pos_width = 300 &
  max_mouse_pos_high = 250
```

Le modèle de la souris elle-même est défini dans la clause `VARIABLES`. L'ensemble `the_post_it_mouse` contient toutes les instances de souris créées à un moment donné (dans la pratique, il n'y en a qu'une à tout moment). Les quatre éléments supplémentaires sont les attributs de la souris (coordonnées et état). L'état est décomposé en deux attributs, l'un interne au modèle reflétant le fait qu'une souris a bien été « informatiquement » créée (`post_it_mouse_creation`) et l'autre qui correspond à l'état externe du bouton de la souris. Le typage est effectué dans la clause `INVARIANT`.

```
VARIABLES
  the_post_it_mouse,
  x_post_it_mouse_pos,
  y_post_it_mouse_pos,
  post_it_mouse_state,
  post_it_mouse_creation
INVARIANT
  the_post_it_mouse   POST_MOUSE &
  x_post_it_mouse_pos   the_post_it_mouse → NAT         &
  y_post_it_mouse_pos   the_post_it_mouse → NAT         &
  post_it_mouse_state   the_post_it_mouse → MOUSE_STATE &
  post_it_mouse_creation   the_post_it_mouse → BOOL     &
  ∀ xx. (xx ∈ the_post_it_mouse ⇒
     (x_post_it_mouse_pos(xx) ∈ 1..max_mouse_pos_width &
      y_post_it_mouse_pos(xx) ∈ 1..max_mouse_pos_high))
```

L'assertion en gras définit une propriété invariante de la souris qui ne doit jamais dépasser les limites de l'écran. Cette propriété doit être garantie quelle que soit l'opération effectuée sur la souris. Ci-dessous, sont décrites les initialisations.

```
INITIALISATION
  the_post_it_mouse         := {}  ||
  x_post_it_mouse_pos       := {}  ||
  y_post_it_mouse_pos       := {}  ||
  post_it_mouse_state       := {}  ||
  post_it_mouse_creation    := {}
```

Nous n'illustrerons que deux des opérations de la souris. La première est celle qui permet de créer un « objet » souris. Elle se contente d'initialiser les différents attributs de la souris.

```
OPERATIONS
 pp ← create_mouse_pos =
  PRE
    the_post_it_mouse ≠ POST_MOUSE
```

```
  THEN
    ANY tt WHERE tt : POST_MOUSE - the_post_it_mouse
      THEN
        pp := tt                                          ||
        the_post_it_mouse := the_post_it_mouse   {tt}     ||
        x_post_it_mouse_pos(tt) := x_mouse_pos_default    ||
        y_post_it_mouse_pos(tt) := y_mouse_pos_default    ||
        post_it_mouse_state(tt) := up                     ||
        post_it_mouse_creation(tt) := TRUE
      END
  END;
```

La seconde sera réutilisée dans la suite de cette section. Elle correspond au déplacement de la souris, bouton enfoncé (« Drag »). La spécification vérifie que la souris est créée, que les nouvelles coordonnées ne violent pas l'invariant, et que le bouton est bien toujours enfoncé. Si toutes ces conditions sont bien remplies, les coordonnées peuvent être mises à jour.

```
 move_mouse_with_drag(pp, aa, bb) =
  PRE
    pp   the_post_it_mouse                        &
    aa   NAT & aa   1..max_mouse_pos_width        &
    bb   NAT & bb   1..max_mouse_pos_high         &
    post_it_mouse_state(pp) = down                &
    post_it_mouse_creation(pp) = TRUE
  THEN
    x_post_it_mouse_pos(pp) := aa          ||
    y_post_it_mouse_pos(pp) := bb
  END;
 move_mouse(pp, aa, bb)= ....;
 mouse_down(pp)= .....;
 mouse_clicked (pp)= .....
END
```

- **Cohérence de la conception**

Nous allons donner un petit aperçu de l'utilisation de l'extension pour montrer comment s'exprime la cohérence de la conception.

Nous allons tout d'abord donner quelques éléments de la machine `POSTIT_VISU_WITHOUT_MOUSE` qui gère la visualisation des notes sans s'occuper de l'aspect interaction.

```
MACHINE POSTIT_VISU_WITHOUT_MOUSE
EXTENDS POSTIT_VISU

CONSTANTS
            ...
PROPERTIES
      max_post_visu=card(POST_VISU)&
      max_post_it_width=300 &
      max_post_it_high=250  &
VARIABLES
...
INVARIANT
      the_post_it_visu   POST_VISU &
      get_new_post   the_post_it_visu → post_new &
      block_state    block_visu &
      post_it_window_status   the_post_it_visu → SCREEN_STATE &
      ∀ xx. ( xx ∈ the_post_it_visu ⇒
             ( x_post_it_position(xx) ∈ 1..max_post_it_width &
               y_post_it_position(xx) ∈ 1..max_post_it_high ) )
```

On définit ici les dimensions maximales de la note, puis, dans l'assertion en gras, on exprime le fait que toutes les fenêtres doivent avoir leur coin supérieur gauche dans l'écran. Là encore, il s'agit d'un invariant, qui doit donc être toujours validé. La suite du code montre la définition partielle de l'opération de déplacement d'une note..

```
OPERATIONS
move_window_position(pp, aa, bb)=
      PRE
             aa   1..max_post_it_width &
             bb   1..max_post_it_high &
             pp   the_post_it_visu   &
             block_state = block_open   &
             ( post_it_window_status (pp) = displayed or
               post_it_window_status (pp) = hidden )
```

Lors de la composition de plusieurs machines, l'utilisation de B garantit la conservation des propriétés. Ainsi, lorsque l'on construit la machine gérant l'interaction en manipulation directe des notes, peut-on identifier des constantes d'une machine à l'autre, et réutiliser les fonctions définies dans d'autres machines, le système obligeant à prouver que les invariants restent bien respectés.

```
MACHINE POSTIT_VISU_WITH_MOUSE
EXTENDS
  POSTIT_MOUSE , POSTIT_VISU_WITHOUT_MOUSE
  …
PROPERTIES
  …
 max_post_it_width = max_mouse_pos_width &
 max_post_it_high = max_mouse_pos_high
 …
move_window_with_mouse(pp, aa, bb)=
PRE
  aa   NAT & aa   1..max_post_it_width &
  bb   NAT & bb   1..max_post_it_high &
  …
  x_post_it_mouse_pos(get_the_mouse(pp))
      x_post_it_pos(get_the_post_it_visu(pp))+5          ..
        x_post_it_pos(get_the_post_it_visu(pp)) +
           x_post_it_window(get_the_post_it_visu(pp)-5            &
  …
THEN
 move_window_pos ( get_the_post_it_visu (pp), aa, bb) ||
 move_mouse_with_drag ( get_the_mouse (pp), aa, bb)
END;
```

Nous verrons dans la section 4.2.3.4 la signification de ces contraintes en termes de propriétés du système interactif.

· **Raffinement**

Le raffinement des spécifications jusqu'au code se fait par étapes successives. Nous avons découpé cette phase en plusieurs sous-phases pour faciliter le travail du « *prover* » automatique. Ainsi, la spécification de la fonction `create_mouse_pos` se raffine-t-elle en introduisant la séquence des opérations. On remarquera que cette opération n'est pas automatique : par exemple, l'affectation du résultat de la fonction doit être postérieure aux autres opérations.

```
pp ← create_mouse_pos =
PRE
  the_post_it_mouse ≠ POST_MOUSE
THEN
 ANY tt WHERE
   tt   POST_MOUSE - the_post_it_mouse
 THEN
  the_post_it_mouse:=the_post_it_mouse   {tt} ;
  x_post_it_mouse_pos(tt) := x_mouse_pos_default ;
  y_post_it_mouse_pos(tt) := y_mouse_pos_default ;
  post_it_mouse_state(tt) := up ;
  post_it_mouse_creation(tt) := TRUE ;
  pp := tt
 END
END
```

La deuxième opération de raffinement consiste à traiter la pré-condition et à définir les blocs du langage. Un bloc `BEGIN/END` est donc créé, et une structure `IF/THEN/END` insérée.

```
pp <--create_mouse_pos=
BEGIN
 IF ( the_post_it_mouse /= POST_MOUSE ) THEN
  ANY tt WHERE
    tt   POST_MOUSE - the_post_it_mouse
  THEN
   the_post_it_mouse := the_post_it_mouse   {tt};
   x_post_it_mouse_pos(tt) := x_mouse_pos_default;
   y_post_it_mouse_pos(tt) := y_mouse_pos_default;
   post_it_mouse_state(tt) := up ;
   post_it_mouse_creation(tt) := TRUE ;
   pp := tt
  END
 END
END;
```

La troisième étape consiste à introduire les variables locales (ici la variable tt de la clause ANY)

```
pp ← create_mouse_pos=
BEGIN
  VAR tt IN
    tt   POST_MOUSE - the_post_it_mouse;
  IF ( the_post_it_mouse ≠ POST_MOUSE )
  THEN
   the_post_it_mouse:=the_post_it_mouse   {tt};
   x_post_it_mouse_pos(tt) := x_mouse_pos_default;
   y_post_it_mouse_pos(tt) := y_mouse_pos_default;
   post_it_mouse_state(tt) := up ;
   post_it_mouse_creation(tt) := TRUE ;
   pp := tt
  END
END ;
```

Ces trois étapes ont été successivement prouvées à l'aide de l'outil « Atelier B » sur l'ensemble de l'exemple. À ce stade, le raffinement obtenu garantit que toutes les propriétés définies dans les spécifications sont encore valides.

· **Implémentation**

Il n'a pas été possible de continuer à raffiner, et surtout d'atteindre le niveau d'implémentation directement avec l'outil B. En effet, la complexité des obligations de preuve générées par l'étape suivante de raffinement, interdisant toute intervention manuelle pour résoudre les preuves non prouvées automatiquement, nous ont obligé à changer de stratégie.

Le passage de structures de données abstraites aux structures de données concrètes a été réalisé à l'aide d'une autre technique formelle (spécifications algébriques). Elle a permis la simplification du développement en B, mais a nécessité la définition dans B de ce raffinement. Le développement n'est donc pas complètement formalisé, mais reste rigoureux dans la mesure où il réutilise les propriétés formellement prouvées. L'exemple suivant permet de comparer la dernière étape de raffinement que nous avons effectuée pour la procédure de déplacement de note avec interaction avec la version Ada finale.

```
Move_mouse_with_drag(pp, aa, bb) =
  BEGIN
   IF ( pp   the_post_it_mouse &
     aa   NAT & aa   1..max_mouse_pos_wide &
     bb   NAT & bb   1..max_mouse_pos_high &
     post_it_mouse_state (pp) = down &
     post_it_mouse_creation (pp) = TRUE )
   THEN
     x_post_it_mouse_pos(pp) := aa ;
     y_post_it_mouse_pos(pp) := bb
  END;
```

Version Ada :

```
procedure move_with_drag ( mm: in out mouse_b ;
            aa: in natural ; bb: in natural ) is
begin
  if ( ( post_it_mouse_creation (mm) = TRUE )
      AND (aa > 0) AND (bb > 0)
      AND (aa <= max_mouse_pos_wide)
      AND (bb <= max_mouse_pos_high)
      AND (post_it_mouse_state(mm) = down))
  then
    x_mouse_pos_set (mm, aa);
    y_mouse_pos_set (mm, bb);
  end if;
end move_with_drag;
```

En ce qui concerne la boîte à outils, seules ont été rétro-conçues les fonctions nécessaires à notre application.

4.2.3.4 B et l'interaction

La méthode B ne permet pas véritablement de « vérifier » des propriétés. Elle permet en revanche de prouver que certaines propriétés sont valides (les invariants).

Ainsi, dans l'exemple que nous avons détaillé, peut-on s'assurer que ni la souris, ni les notes ne peuvent se trouver à un moment donné hors de l'écran. En ce qui concerne la souris, on peut

considérer qu'il s'agit là d'une propriété de base de la boîte à outils qu'il n'aurait pas été nécessaire de démontrer. Cependant, la conception modulaire de B permet d'obtenir des résultats plus originaux.

Ainsi, en « étendant » les deux machines `POST_IT_MOUSE` et `VISU_WITHOUT_MOUSE` pour concevoir la machine `VISU_WITH_MOUSE` nous avons créé une nouvelle condition (la zone de manipulation directe de la note ne doit jamais sortir de l'écran), spécifique au mode d'interaction choisi. L'intérêt de B est double. D'une part, il permet d'introduire ces contraintes indépendamment les unes des autres, ce qui est un atout supplémentaire pour la conception et la réutilisation. D'autre part, il assure la cohérence entre les différentes propriétés ; dans ce cas précis, le « *prover* » vérifiera que toutes les contraintes exprimées sont compatibles, ce qui est vrai dans notre cas car la contrainte sur la zone de déplacement est moins forte que celle exprimée dans la machine `VISU_WITHOUT_MOUSE`.

Un autre grand avantage de B sur la majorité des méthodes utilisées dans la littérature réside dans la nature de ce qui est formellement démontré par le système. Nous avons pu démontrer les propriétés du système sur ses spécifications, mais les étapes de raffinement réalisées en B assurent que ces propriétés sont conservées jusqu'à la dernière étape du raffinement. L'utilisation de schémas de programmes prouvés permet de garantir que l'application elle-même répond à ses spécifications, et respecte les propriétés exprimées dès le début du développement. Dans la plupart des cas, les vérifications effectuées sur les systèmes interactifs portent sur une modélisation du système, qui n'est pas ensuite raffinée en un système réel. La conformité du système réalisé par rapport au modèle n'est pas démontrée.

Bien que ce travail soit limité à un cas d'étude, nous avons pu montrer que la méthode B permettait d'assurer des propriétés dans les trois classes de propriétés identifiées par [Campos & Harrison 1997], à savoir la visibilité (L'état du système est-il perceptible ?) l'atteignabilité ou *reachability* (Peut-on toujours agir sur une note ?) et la sûreté ou *reliability* (Les invariants du noyau fonctionnel sont-ils préservés ?).

L'étude présentée ici a été détaillée dans [Aït-Ameur, Girard, & Jambon 1998b] et a fait l'objet de deux articles publiés [Aït-Ameur, Girard, & Jambon 1998a ; Aït-Ameur, Girard, & Jambon 1998c]). Le raffinement ainsi qu'un article de synthèse sont actuellement en phase de soumission.

4.3 Perspectives

Compte tenu de leur caractère globalement préliminaire, les travaux que nous avons amorcés ouvrent sur de nombreuses perspectives.

4.3.1 Vérification des systèmes

Les travaux menés avec EXPRESS sur l'expression du dialogue dans le modèle des interacteurs hiérarchisés peuvent être étendus pour élaborer des outils de vérification du dialogue. Une transformation des dialogues utilisés dans l'application sous forme de fichiers physiques EXPRESS permettrait ainsi la conservation des propriétés exprimées par le modèle des interacteurs. De plus, l'outil ECCO peut être utilisé pour animer des instances de dialogues dans un but de prototypage.

4.3.2 Extension de la validation

Les travaux menés sur la validation des systèmes interactifs portaient sur une application bien particulière utilisant le modèle des interacteurs hiérarchisés. Comme nous l'avons évoqué dans les perspectives du chapitre précédent, il nous paraît souhaitable de connecter ce travail à celui des différents auteurs travaillant dans le domaine de la modélisation des tâches de l'utilisateur (« Task-Based Modelling »). L'utilisation de formats existants garantirait une plus grande portée des résultats. L'extension de la méthode de validation à des modèles de dialogue différents serait également d'un grand intérêt.

L'intégration de notre méthode de validation à un environnement permettant de lier modèle de tâche et implémentation (comme nous l'avons envisagé en 3.3.2) complèterait l'approche, et autoriserait une couverture complète des phases de conception, de validation, et de réalisation, incluant la phase d'évaluation.

4.3.3 Utilisation de B

Nous avons appliqué la méthode B à un cas d'école bien spécifique. Les ouvertures de ce travail sont nombreuses.

Le recensement de l'ensemble des propriétés validables en B, et la définition de véritables schémas de spécification adaptés aux problèmes de l'interaction, équivalents aux « design patterns » du génie logiciel, est un sujet qui nous semble très prometteur. [van Weilie, van der Veer, & Eliëns 1999] définit ainsi un modèle de dialogue incluant de nombreuses propriétés qui ne sont généralement évaluables qu'à la fin de la conception. La traduction de ces propriétés en B serait ainsi un premier pas.

La définition d'un outil permettant d'effectuer une corrélation entre les propriétés et contraintes graphiques, définies par exemple dans les constructeurs d'interface, et les invariants exprimables dans les machines B, est de nature à favoriser l'utilisation de la méthode B dans les interfaces. L'application de techniques issues de la PsE/PbD à la définition de propriétés en B constitue une voie possible

La définition d'une véritable méthode de développement intégrant B, qui utilise un modèle de tâches en entrée, est également un problème digne d'intérêt. Les derniers travaux sur les modèles de tâche évoluent en effet vers une modélisation à base de pré et post-conditions, dont l'expression formelle en B est parfaitement réalisable. Il semble intéressant d'étudier et de recenser les mécanismes permettant de définir des modèles formels à partir des modèles informels des différents auteurs. Ceci constituerait un premier pas vers une méthode de développement rigoureuse.

Enfin, l'application de la méthode B à des cas concrets correspondant aux domaines qui nécessitent une grande sûreté, comme nous l'avons amorcé dans [Aït-Ameur, et al. 1998] peut également être envisagée.

4.3.4 Intégration de méthodes

Comme nous l'avons dit, B ne constitue en rien la méthode universelle. De nombreuses propriétés ne peuvent être vérifiées en B, ou bien demandent un effort considérable. En particulier, les propriétés entièrement dynamiques sont difficiles à assurer. La combinaison de méthodes est ainsi un travail que nous envisageons, que ce soit entre méthodes que nous avons utilisées (B, EXPRESS) ou entre méthodes utilisées par les auteurs (Esterel, Lustre, Réseaux de Petri, etc.).

Chapitre 5
Programme de recherches

L'utilisateur « éclairé » est de plus en plus demandeur de possibilités « d'adaptations » de son système. Parallèlement, l'ingénierie des systèmes interactifs tend à fournir de plus en plus de méthodes et outils permettant de raisonner sur les systèmes. La réunion de ces deux tendances a-t-elle un sens ? Au premier abord, la réponse est non. La génération automatique de systèmes interactifs à partir de modèles divers introduit une barrière interdisant tout retour en arrière comme celui qui consisterait à vouloir modifier des modèles après avoir adapté la présentation du système. L'adaptation du système par l'utilisateur qui, si on introduit des techniques de PsE/PbD, peut être extrêmement profonde, ne peut dans ce schéma être répercutée au niveau des modèles ayant servi de base à la génération. Quant à l'intégration des méthodes formelles dans le schéma de conception des systèmes interactifs, elle aboutit à un éloignement encore plus grand de l'espace de conception de l'espace de l'utilisateur. Si les approches actuelles « centrées-utilisateur » (*user-centred design*) mettent en avant le rôle fondamental du concepteur dans la définition et l'évaluation au plus tôt des systèmes interactifs, elles ne mettent en aucun cas l'utilisateur au centre de la conception ; il en est au contraire évincé par l'utilisation de techniques, tant de programmation que de modélisation, qui lui sont inaccessibles.

Nous avons montré avec GIPSE que cette étape de génération pouvait être supprimée. Nous avons même expérimenté la possibilité de modifier interactivement tous les composants de l'interface utilisateur, bien au-delà de la seule couche de présentation. Notre projet de recherche consiste à replacer l'utilisateur **au centre du processus de conception** des systèmes interactifs. À partir d'outils de base et de méthodes accessibles interactivement, il doit être en mesure de réaliser ses besoins, tout en vérifiant qu'ils sont conformes à des règles définies généralement, ou même qu'il aura lui-même définies. Cet objectif nécessite deux démarches antagonistes. La première consiste à permettre à l'utilisateur de construire son application à partir d'éléments existants, en les adaptant à son propre usage. La seconde consiste à lui permettre de demeurer conforme à la logique et aux règles gouvernant l'application.

5.1 Du formalisme à l'utilisateur

GIPSE permet de construire une application en partant d'un noyau fonctionnel existant. La synthèse des résultats que nous avons obtenus, alliée aux développements actuellement en cours au sein du laboratoire ([Texier & Guittet 1999], thèse de F. Depaulis) devraient nous permettre de fournir à l'utilisateur des possibilités beaucoup plus étendues dans la création d'une nouvelle application à partir d'un noyau existant.

Une première étape consiste à définir une application interactive dont le noyau fonctionnel serait connu par une interface complètement formalisée (par exemple en B). Le travail préliminaire de définition des besoins des interfaces par rapport au noyau fonctionnel de [Fekete 1996a] apporte ici un point de départ intéressant. L'utilisation de boîtes à outils spécialisées dans la manipulation directe comme Amulet [Myers, et al. 1997] devrait permettre une mise en œuvre rapide.

Une deuxième étape concerne la richesse des dialogues. Nous avons étudié plusieurs aspects concernant les dialogues structurés et leur amélioration, tant sur le plan de la facilité de spécification que sur celui de la facilité d'utilisation. Cependant, l'intégration de plusieurs formes de dialogue demeure encore incomplète. L'amélioration de cet aspect, mais aussi la mise à disposition de l'utilisateur de possibilités plus grandes pour définir le dialogue qui lui convient, sont des objectifs à atteindre. Pour y parvenir, nous envisageons d'étendre nos investigations à des domaines autres que la CAO, comme nous l'avons commencé avec le travail de F. Depaulis. Les collecticiels, mais

également la modélisation 3D et l'interaction en réalité virtuelle, ainsi que l'interaction avec des documents sur le web sont des voies susceptibles d'être explorées. Ceci devrait nous conduire à la définition d'un multi-contrôleur de dialogue pour applications graphiques interactives adaptable à des situations très diverses.

5.2 De l'utilisateur au formalisme

Dans un premier temps, la barrière de la génération que nous avons décrite en tout début de chapitre doit être contournée. En allant plus loin, on doit même pouvoir aller indifféremment des modèles et formalismes à l'application et inversement. Cette approche, que l'on trouve sous une forme limitée aux composants de la boîte à outils dans les travaux de Lecolinet [Lecolinet 1996], doit être étendue à l'ensemble des modèles utilisés. Là encore, nous avons engagé des travaux en ce sens avec GIPSE. La démarche mérite cependant d'être systématisée, et surtout étendue à des aspects des systèmes interactifs autres que le seul dialogue homme-machine. Au lieu d'une simple adaptation du système interactif, l'utilisateur sera en mesure de modifier profondément son système, voire de l'étendre, tout en restant cohérent avec le système.

Dans un deuxième temps, il convient de s'assurer que les modifications et ajouts effectués ne violent pas les propriétés du système. La vérification interactive de propriétés sur le système modifié doit ainsi être possible. Les travaux effectués dans le cadre du DEA de Y. Boisdron peuvent ainsi servir de base. La définition d'outils en relation avec les travaux des différents auteurs sur les modèles de tâche devrait permettre de faire le lien entre la conception de ces modèles et les vérifications possibles sur le modèle et surtout l'application finale. Il est nécessaire pour cela de dépasser le stade de la simple tâche « utilisateur », souvent abstraite et indépendante des tâches d'articulation, pour atteindre un niveau proche de l'interaction finale.

Cependant, pour atteindre ces différents objectifs, il est nécessaire d'assister l'utilisateur. Une première voie consiste à lui fournir des outils intelligents qui l'aident à réaliser sa tâche. Cette approche présente deux inconvénients : le premier est qu'elle est nécessairement limitée dans son spectre de problèmes pouvant être traités, le second est qu'elle incite l'utilisateur à se décharger sur la machine ; s'il ne sait pas comment faire, l'assistant le saura certainement, et peu importe comment… Notre démarche a toujours suivi une deuxième voie, celle de l'utilisation explicite des mécanismes nécessaires à l'accomplissement de la tâche. Cependant, cette solution n'empêche pas de rechercher des moyens d'apporter une assistance à l'utilisateur. L'assistance ne concerne pas la tâche elle-même, mais les mécanismes susceptibles d'être utilisés pour réaliser la tâche. Appliqué à notre problème global, on peut le décliner de deux manières : par rapport aux mécanismes de programmation, et par rapport aux mécanismes de spécification formelle.

Trouver une manière simple pour apprendre la programmation a toujours été un défi pour les informaticiens. La PsE/PbD a apporté sa contribution au problème avec des systèmes comme Tinker [Lieberman 1993]. Mais il faut aller plus loin dans l'aide à l'apprentissage de l'algorithmique ou de la programmation événementielle. De plus, la PsE/PbD ne s'est attaquée qu'à des problèmes concernant le « Programming in the small », dépassant rarement le niveau du sous-programme dans les éléments de structuration du code. La notion de module ou de classe n'est pas traitée. Si nous désirons permettre à l'utilisateur d'ajouter de nouvelles classes dans son application, il faut qu'il en appréhende toutes les facettes. Comment lui rendre accessibles simplement ? Les méthodes orientées-objets utilisent abondamment les notations graphiques. Sont-elles naturelles pour un utilisateur ? Dans le domaine plus spécifique de l'interaction homme-machine, la concrétisation d'une conception d'application passe par des choix comme celui des techniques et des formes d'interaction. Comment mettre ces notions à la portée de l'utilisateur, qui les manipule pourtant dès qu'il touche à un système informatique ?

Le deuxième problème concerne l'utilisation des spécifications formelles. Si nous souhaitons voir l'utilisateur vérifier et valider son système, il devra aller au-delà de l'utilisation d'outils automatiques de vérification de complétude ou d'atteignabilité. En règle générale, il sera capable d'exprimer, le plus souvent graphiquement, des règles concernant son interface. Par exemple, « *ces*

objets doivent être alignés » ou « *tel bouton doit être toujours visible* ». Mais traduire ces règles en propriétés dans un langage de spécifications est actuellement largement au-delà de la portée de l'utilisateur. En quoi la PsE/PbD peut-elle aider à comprendre les mécanismes des langages de spécification formelles et à réaliser ces mêmes spécifications ? Il nous semble que la bonne décomposition des concepts de la méthode B (attributs, invariants, etc.) est de nature à permettre des travaux en ce sens.

5.3 Mettre l'utilisateur au centre du processus de conception/réalisation

Afin de mettre réellement l'utilisateur au centre du processus de conception, il convient de lui permettre d'appréhender les concepts nécessaires à la conception, et de lui donner les outils nécessaires à la réalisation. C'est pourquoi nous avons intitulé cette section « Mettre l'utilisateur au centre du processus de conception/réalisation ».

Le développement des IHM fait largement appel aux cycles de développement incrémentaux. L'alternance des phases de conception et de prototypage est la règle. Donner les moyens à l'utilisateur de s'intégrer plus profondément dans ce cycle est notre objectif. À l'aide d'outils totalement interactifs, permettant d'interagir directement avec les modèles sous-jacents mais également avec l'exemple en cours d'exécution, nous pensons que l'utilisateur doit être en mesure d'aller au-delà de simples modifications superficielles, concernant les aspects de pure présentation par exemple.

Il n'est cependant pas question de retirer les autres participants du cycle. Ceux-ci, qu'ils soient psychologues, ergonomes ou informaticiens, seront toujours nécessaires à la résolution de problèmes durs, ponctuels par rapport à l'application. Mais l'utilisateur sera à même d'intervenir directement, sans mettre en péril l'équilibre global de l'application.

Bibliographie

1. [Abowd, Wang, & Monk 1995] Abowd G.D., Wang H.-M., & Monk A.F. A Formal Technique for Automated Dialogue Development. *DIS'95, Design of Interactive Systems*, Eds. G.M. Olson & S. Schuon, Pub. ACM Press, *Ann Arbor, Michigan*, August 23-25 1995, pp. 219-226.
2. [Abrial 1996] Abrial J.-R. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996, 779p.
3. [Accott et al. 1997] Accott J., Chatty S., Maury S., & Palanque P. Formal transducers: Models of devices and building bricks for the design of highly interactive systems. *Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'97)*, Eds. M.D. Harrison & J.C. Torres, Ser. SpringerComputerScience, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Granada, Spain*, 4-6 June 1997, pp. 143-160.
4. [Aït Ameur et al. 1995] Aït Ameur Y., Besnard F., Girard P., Pierra G., & Potier J.-C. Specification and Metaprogramming in the EXPRESS Language. *Intern. Conference on Software Engineering and Knowledge Engineering*, Pub. IEEE, ACM, *Rockville, USA*, June 1995, pp. 181-189.
5. [Aït-Ameur et al. 1998] Aït-Ameur Y., Cottet F., Girard P., & Jambon F. Complete Formal Analysis of Timeliness in En-Route Air-Traffic Control User Interfaces. *Workshop on Design User Interface for Safety Critical Systems (CHI'98)*, Pub. ACM Press, *Los Angeles (CA), USA*, 18-23 April 1998.
6. [Aït-Ameur, Girard, & Jambon 1998a] Aït-Ameur Y., Girard P., & Jambon F. A Uniform approach for the Specification and Design of Interactive Systems: the B method. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98)*, Eds. P. Markopoulos & P. Johnson, *Abingdon, UK*, vol. Proceedings, 3-5 June 1998a, pp. 333-352.
7. [Aït-Ameur, Girard, & Jambon 1998b] Aït-Ameur Y., Girard P., & Jambon F. *Using the B formal approach for incremental specification design of interactive systems*. Laboratoire d'Informatique Scientifique et Industrielle (LISI/ENSMA), February 1998b, Research report 98-001.
8. [Aït-Ameur, Girard, & Jambon 1998c] Aït-Ameur Y., Girard P., & Jambon F. Using the B formal approach for incremental specification design of interactive systems. *Engineering for Human-Computer Interaction (IFIP TC2/TC13 WG2.7/WG13.4 Working Conference EHCI'98)*, Eds. S. Chatty & P. Dewan, Pub. Kluwer Academic Publishers, *Heraklion, Greece*, vol. 22, 14-18 September 1998c, pp. 91-108.
9. [Al-Qaimari & McRostie 1999] Al-Qaimari G. & McRostie D. KALDI: a computer-based usability engineering tool for supporting testing and analysis of human-computer interaction. *Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)*, Eds. J. Vanderdonkt & A. Puerta, Pub. Kluwer Academic Publishers, *Louvain-la-neuve, Belgique*, 21-23 October 1999, pp. 337-355.
10. [Amai 1994] Amai N. *Réalisation d'une interface d'exploration d'une réalité virtuelle par synchronisation de plusieurs niveaux d'interacteurs*. Rapport de DEA : LISI, ENSMA, Poitiers, Futuroscope, 1994, 48p.
11. [Ballardin, Mancini, & Paternò 1999] Ballardin G., Mancini C., & Paternò F. Computer-Aided Analysis of Cooperative Applications. *Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)*, Eds. J. Vanderdonkt & A. Puerta, Pub. Kluwer Academic Publishers, *Louvain-la-neuve, Belgique*, 21-23 October 1999, pp. 257-270.
12. [Balzert 1995] Balzert H. From OOA to GUI : The JANUS-System. *IFIP TC13 Human-Computer Interaction (INTERACT'95)*, Eds. K. Nordby, P.H. Helmersen, D.J. Gilmore & S.A. Arnesen, Pub. Chapmann & Hall, *Lillehammer, Norway*, 27-29 June 1995, pp. 319-324.
13. [Balzert et al. 1996] Balzert H., Hofmann F., Kruschinski V., & Niemann C. The JANUS Application Development Environment-Generating more than the User Interface. *Computer-Aided Design of User Interface (CADUI'96)*, Ed. J. Vanderdonckt, Pub. Presse Universitaire de Namur, *Namur, Belgium*, 5-7 June 1996, pp. 183-206.
14. [Barthet 1988] Barthet M.-F. *Logiciels Interactifs et Ergonomie, Modèles et méthodes de conception.* Paris : Dunod Informatique, 1988, 219p.
15. [Bass & Coutaz 1991] Bass L. & Coutaz J. *Developing Software for the User Interface*. Addison-Wesley, 1991, 256p.
16. [Bass et al. 1992] Bass L., Faneuf R., Little R., Mayer N., Pellegrino B., Reed S., Seacord R., Sheppard S., & Szczur M.R. A Metamodel for the Runtime Architecture of an Interactive System. *SIGCHI Bulletin*, vol. 24, n° 1, 1992, pp. 32-37.
17. [Bauer 1979] Bauer M.A. Programming by Examples. *Articial Intelligence*, vol. 12, 1979, pp. 1-21.

18. [Beaudouin-Lafon, Berteaud, & Chatty 1990] Beaudouin-Lafon M., Berteaud Y., & Chatty S. Créer des applications à manipulation directe avec Xtv. *Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'90), Biarritz,* 1990, pp. 78-88.
19. [Bhavnami & John 1996] Bhavnami S.K. & John B.E. Exploring the Unrealized Potential of Computer Aided Drafting. *Human Factors in Computing Systems (CHI'96),* Ed. M. Tauber, Pub. ACM/SIGCHI, *Vancouver, Canada,* 13-18 April 1996, pp. 232-239.
20. [Biere, Bomsdorf, & Szwillus 1999] Biere M., Bomsdorf B., & Szwillus G. The Visual Task Model Builder. *Third Conference on Computer-Aided Design of User Interfaces (CADUI'99),* Eds. J. Vanderdonkt & A. Puerta, Pub. Kluwer Academic Publishers, *Louvain-la-neuve, Belgique,* 21-23 October 1999, pp. 245-256.
21. [Bodart et al. 1995a] Bodart F., Hennebert A.-M., Leheureux J.-M., Provot I., B. Sacré, & Vanderdonckt J. Towards a systematic building of software Architectures : the Trident Methodological Guide. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'95),* Eds. P. Palanque & R. Bastide, Pub. Springer-Verlag/Wien, *Bonas, France,* 1995a, pp. 262-278.
22. [Bodart et al. 1995b] Bodart F., Hennebert A.-M., Leheureux J.-M., Provot I., & Vanderdonckt J. A Model-Based Approach to Presentation: a Continuum from Task Analysis to Prototype. *Interactive Systems: Design, Specification, and Verification (DSV-IS'94),* Ed. F. Paternò, Ser. Focus on Computer Graphics, Ed. Ser. Eurographics, Pub. Springer, *Bocca di Magra, Italy,* 8-10 June 1995b, pp. 77-94.
23. [Borning 1981] Borning A. The Programming Langugage Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, vol. 3, n° 4, 1981, pp. 353-387.
24. [Bouazza 1995a] Bouazza M. *La norme STEP*. Paris : Hermès, 1995a, 139p.
25. [Bouazza 1995b] Bouazza M. *Le langage EXPRESS*. Paris : Hermès, 1995b, 304p.
26. [Bouma et al. 1995] Bouma W., Fudos I., Hoffmann C., Cai J., & Paige R. Geometric Constraint Solver. *Computer Aided Design*, vol. 27, n° 6, 1995, pp. 487-501.
27. [Brun 1997] Brun P. *XTL: a temporal logic for the formal development of interactive systems*. *in* Formal Methods for Human-Computer Interaction, Eds. P. Palanque & F. Paternò, Springer-Verlag, 1997, pp. 121-139.
28. [Brun & Beaudouin-Lafon 1995] Brun P. & Beaudouin-Lafon M. A taxonomy and evaluation of formalisms for the specification of interactive systems. *Conference of the British Human-Computer Interaction Group (HCI'95),* Eds. M.A.R. Kirby, A.J. Dix & J.E. Finlay, Ser. People and Computers, Pub. Cambridge University Press, *University of Huddersfeild, UK*, vol. X, August 1995, pp. 197-212.
29. [Bumbulis et al. 1996] Bumbulis P., Alencar P.S.C., Cowan D.D., & Lucena C.J.P. Validating properties of component-based graphical user interfaces. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96),* Eds. F. Bodart & J. Vanderdonckt, Pub. Springer-Verlag, *Namur, Belgium*, 5-7 June 1996, pp. 347-365.
30. [Cabrera, Torres, & Gea 1999] Cabrera M., Torres J.C., & Gea M. Towards User Interfaces Prototyping from Algebraic Specification. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99), Universidade do Minho, Braga, Portugal*, vol. Conference proceedings, 2-4 June 1999, pp. 55-69.
31. [Campos & Harrison 1997] Campos J.C. & Harrison M.D. Formally Verifying Interactive Systems: A Review. *Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'97),* Eds. M.D. Harrison & J.C. Torres, Ser. SpringerComputerScience, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Granada, Spain,* 4-6 June 1997, pp. 109-124.
32. [Campos & Harrison 1999] Campos J.C. & Harrison M.D. Using automated reasoning in the design of an audio-visual communication system. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99), Universidade do Minho, Braga, Portugal*, vol. Conference proceedings, 2-4 June 1999, pp. 161-182.
33. [Card, Moran, & Newell 1983] Card S., Moran T., & Newell A. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983, 280p.
34. [Carr 1997] Carr D. *Interaction Object Graphs: An Executable Graphical Notation for Specifying User Interfaces*. *in* Formal Methods for Human-Computer Interaction, Eds. P. Palanque & F. Paternò, Springer-Verlag, 1997, pp. 141-156.
35. [Castells & Szekely 1999] Castells P. & Szekely P. Presentation Models by Example. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99), Universidade do Minho, Braga, Portugal*, vol. Conference proceedings, 2-4 June 1999, pp. 89-105.
36. [Chang 1986] Chang S.-K. *Visual languages and iconic languages*. *in* Visual languages, Eds. S.-K. Chang, T. Ichikawa & P. Ligomenides, New-York : Plenum Press, 1986, pp. 1-7.
37. [Chatty 1992] Chatty S. *La construction d'interfaces homme-machine animées*. Doctorat d'Université (PhD Thesis) : LRI, Université Paris-Sud, Orsay, 1992, 196p.

38. [Chatty, Girard, & Sire 1996] Chatty S., Girard P., & Sire S. Vers un support multimédia à la collaboration directe. *Technique et Science Informatique*, vol. 15, n° 9, 1996, pp. 1259-1286.
39. [Chen 1976] Chen P.P.-S. The Entity-Relationship Model. Toward a Unified View of Data. *ACM Transactions on Database Systems*, vol. 1, n° 1, 1976, pp. 9-36.
40. [Coutaz 1990] Coutaz J. *Interfaces Homme-Ordinateur, Conception et Réalisation*. Paris : Dunod Informatique, 1990, 455p.
41. [Cugini, Folini, & Vicini 1988] Cugini U., Folini F., & Vicini I. A Procedural System for Definition and Storage of Technical drawings in Parametric Form. *EUROGRAPHICS'88*, Pub. Eurographics, 1988, pp. 183-196.
42. [Cypher 1991] Cypher A. Eager: Programming Repetitive Tasks by Example. *Human Factors in Computing Systems (CHI'91)*, Pub. ACM/SIGCHI, *New Orleans, Louisiana*, 27 April - 2 May 1991, pp. 33-39.
43. [Cypher 1993a] Cypher A. *Eager : Programming Repetitive Tasks by Demonstration*. *in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993a, pp. 205-217.
44. [Cypher 1993b] Cypher A. *Introduction: Bringing Programming to End Users*. *in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993b, pp. 1-11.
45. [Cypher 1993c] *Watch What I Do: Programming by Demonstration*. Ed. Cypher A., Cambridge, Massachusetts : The MIT Press, 1993c, 604p.
46. [Cypher, Kosbie, & Maulsby 1993] Cypher A., Kosbie D.S., & Maulsby D. *Characterizing PBD Systems*. *in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993, pp. 467-484.
47. [Cypher & Smith 1995] Cypher A. & Smith D.C. KidSim: End User Programming of Simulations. *Human Factors in Computing Systems (CHI'95)*, Pub. ACM/SIGCHI, *Denver, Colorado*, 7-11 May 1995, pp. 27-36.
48. [d'Ausbourg 1998] d'Ausbourg B. Using Model Checking for the Automatic Validation of User Interface Systems. *Eurographics Workshop on Design, Specification and Validation of Interactive Systems (DSV-IS'98)*, Eds. P. Markopoulos & P. Johnson, Pub. SpringerWienNewYork, *Abingdon, UK*, 3-5 June 1998, pp. 242-260.
49. [d'Ausbourg & Cazin 1999] d'Ausbourg B. & Cazin J. Using TRIO Specifications to Generate Test Cases for an Interactive System. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99), Universidade do Minho, Braga, Portugal*, vol. Conference proceedings, 2-4 June 1999, pp. 143-160.
50. [d'Ausbourg, Durrieu, & Roché 1996] d'Ausbourg B., Durrieu G., & Roché P. Deriving a Formal Model of an Interactive System from its UIL Description in order to Verify and Test its Behaviour. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96)*, Eds. F. Bodart & J. Vanderdonckt, Ser. SpringerComputerScience, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Namur, Belgium*, 5-7 June 1996, pp. 105-122.
51. [de Bruin, Bowman, & van den Bos 1993] de Bruin H., Bowman P., & van den Bos J. DIGIS, a Graphical User Interface Environment for non programmer. *Computer Graphics Forum (Eurographics)*, vol. 12, n° 3, 1993, pp. C13-C24.
52. [de Haan 1995] de Haan G. *ETAG-based Design: User Interface Design as user Mental Model Design*. *in* Critical Issues in User Interface Engineering, Eds. P. Palanque & D. Benyon, London : Springer-Verlag, 1995, pp. 81-92.
53. [Dix et al. 1998] Dix A., Finlay J., Abowd G., & Beale R. *Human-Computer Interaction*. Prentice Hall, 1998, 638p.
54. [Dufourd 1990] Dufourd J.-F. Programmtion et résolution de problèmes de construction géométrique. *Bigre*, vol. 70, 1990, pp. 136-147.
55. [Duke et al. 1994] Duke D.J., Faconti G.P., Harrison M.D., & Paternò F. Unifying views of interactors. *International Workshop on Advanced Visual Interfaces*, Pub. ACM Press, 1994, pp. 143-152.
56. [Duke & Harrison 1993] Duke D.J. & Harrison M.D. Abstract Interaction Objects. *Computer Graphics Forum*, vol. 12, n° 3, 1993, pp. 25-36.
57. [Duval 1994] Duval T. Modèles d'architecture pour les applications graphiques interactives : la famille Seeheim. *Revue Internationale de CFAO et d'Infographie*, vol. 9, n° 4, 1994, pp. 529-555.
58. [EXPRESS 1994] EXPRESS. *The EXPRESS language reference manual*. ISO, 1994 ISO 10303-11.
59. [Faconti & Duke 1996] Faconti G.P. & Duke D.J. Device Models. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96)*, Eds. F. Bodart & J. Vanderdonckt, Ser. SpringerComputerScience, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Namur, Belgium*, 5-7 June 1996, pp. 73-91.

60. [Farenc & Palanque 1999] Farenc C. & Palanque P. A Generic Framework based on Ergonomics Rules for Computer-Aided Design of User Interfaces. *Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)*, Eds. J. Vanderdonkt & A. Puerta, Pub. Kluwer Academic Publishers, *Louvain-la-neuve, Belgique*, 21-23 October 1999, pp. 281-292.
61. [Fekete 1996a] Fekete J.-D. Les trois services du noyau sémantique indispensables à l'IHM. *Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'96)*, Ed. J. Coutaz, Pub. Cépaduès, *Grenoble*, 16-18 Septembre 1996a, pp. 45-50.
62. [Fekete 1996b] Fekete J.-D. *Un modèle multicouche pour la construction d'applications graphiques interactives*. Doctorat d'Université (PhD Thesis) : LRI, Université Paris-Sud, Orsay, 1996b, 203p.
63. [Fields, Merriam, & Dearden 1997] Fields B., Merriam N., & Dearden A. DMVIS: Design, Modelling and Validation of Interactive Systems. *Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'97)*, Eds. M.D. Harrison & J.C. Torres, Ser. SpringerComputerSeries, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Granada, Spain*, 4-6 June 1997, pp. 29-44.
64. [Foley et al. 1990] Foley, Dam v., Feiner, & Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley Publishing Comp., 1990, 1174p.
65. [Foley & Sukaviriya 1994] Foley J.D. & Sukaviriya P.N. History, Results and Bibliography of the User Interface Design Environment (UIDE), an Early Model-Based System for User Interface Designe and Implementations. *Interactive Systems: Design, Specification, and Verification (DSV-IS'94)*, Ed. F. Paternò, Pub. Springer, *Bocca di Magra, Italy*, 8-10 June 1994, pp. 3-14.
66. [Forbrig 1999] Forbrig P. Task- and Object-Oriented Development of Interactive Systems - How many models are necessary? *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99), Universidade do Minho, Braga, Portugal*, vol. Conference proceedings, 2-4 June 1999, pp. 223-238.
67. [Gamboa & Scapin 1997] Gamboa R.F. & Scapin D.L. Editing MAD* task description for specifying user interfaces, at both semantic and presentation levels. *Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'97)*, Eds. M.D. Harrison & J.C. Torres, Ser. SpringerComputerScience, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Granada, Spain*, 4-6 June 1997, pp. 193-208.
68. [Gardan 1991] Gardan Y. *La CFAO introduction, techniques et mise en oeuvre*. Paris : Hermès, 1991, 418p.
69. [Gardan et al. 1988] Gardan Y., Jung J.-P., Kolopp J.-N., Minich C., & Totino W. Une approche nouvelle de la convivialité dans un système de CAO : les principes de dialogue dans SACADO. *MICAD*, Ed. Hermès, *Paris, 21-25 Mars*, 1988, pp. 281-296.
70. [Gardan, Jung, & Martin 1993] Gardan Y., Jung J.-P., & Martin B. An End-User oriented approach to design man-machine interface for CAD/CAM. *IEEE Internationel Conference on Systems, Man and Cybernetics, Le Touquet, France, 17-20 Octobre 1993*, 1993, pp. 525-530.
71. [Gaudel et al. 1996] Gaudel M.-C., Marre B., Schlienger F., & Bernot G. *Précis de génie logiciel*. Paris : Masson, 1996, 142p.
72. [Girard 1992] Girard P. *Environnement de Programmation pour Non-Programmeur et Paramétrage en Conception Assistée par Ordinateur : le système LIKE*. Doctorat d'Université (PhD Thesis) : LISI/ENSMA, Université de Poitiers, 1992, 195p.
73. [Girard et al. 1997] Girard P., Patry G., Pierra G., & Potier J.-C. Deux exemples d'utilisation de la Programmation par Démonstration en Conception Assistée par Ordinateur. *Revue Internationale de CFAO et d'informatique graphique*, vol. 12, n° 1-2, 1997, pp. 169-188.
74. [Girard & Pierra 1990] Girard P. & Pierra G. End User Programming Environments : Interactive Programming-On-Example in CAD Parametric Design. *EUROGRAPHICS'90*, Pub. Eurographics, *Montreux*, 3-7 Sept 1990, pp. 261-274.
75. [Girard & Pierra 1993] Girard P. & Pierra G. Command Recording versus Parametric and Variational Systems, and old/new third way of parametrizing CAD models by End Users. *COMPEURO'93*, Pub. IEEE Comp. Society Press, *Evry, France*, Mai 1993, pp. 194-200.
76. [Girard & Pierra 1995a] Girard P. & Pierra G. Programming by Demonstration, a user-oriented programming paradigm for graphic systems. *Eurographics Workshop on Object-Oriented Programming Paradigm*, Eds. R. Veltkamp & E. Blake, Pub. Eurographics, *Maastricht*, 1-2 Sep. 1995a, pp. 217-232.
77. [Girard & Pierra 1995b] Girard P. & Pierra G. Structures de contrôle générales en Programmation par Démonstration. *Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'95)*, Ed. P. Palanque, Pub. Cépaduès, *Toulouse*, 11-13 Octobre 1995b, pp. 61-68.
78. [Girard, Pierra, & Guittet 1995] Girard P., Pierra G., & Guittet L. Les interacteurs hiérarchisés : une architecture orientée tâches pour la conception des dialogues. *Revue d'Automatique et de Productique Appliquée (RAPA)*, vol. 8, n° 2-3, 1995, pp. 235-240.

79. [Girard, Pierra, & Potier 1997] Girard P., Pierra G., & Potier J.-C. Customizing by Demonstration Generic Systems to Specific Tasks. *UI4ALL, 3rd ERCIM Workshop on User Interfaces for All*, Ed. N. Carbonell, Pub. ERCIM & INRIA Lorraine, *Ortrott, France*, 3-4 November 1997, pp. 189-196.
80. [Girard & Potier 1995] Girard P. & Potier J.-C. L'activité coopérative en conception technique : quelques règles. *Quatrième table ronde francophone sur la conception, 01Design'95, Autrans, France*, 11-13 Janvier 1995, pp. 215-229.
81. [Girard, Potier, & Pierra 1996] Girard P., Potier J.-C., & Pierra G. *EBP : Example-Based programming in parametrics*, Grenoble, Cassette Video IHM'96, 1996.
82. [Glinert 1987] Glinert E. Out of Flatland : towards 3D Visual Programming. *IEEE FJCC'87, Dallas, Texas, 25-29 Oct*, 1987, pp. 292-299.
83. [Glinert 1990] *Visual programming environments: paradigms and systems*. Ed. Glinert E.P., Los Alamitos, California : IEEE Computer Society Press, 1990, 660p.
84. [Gray, England, & McGowan 1994] Gray P., England D., & McGowan S. *XUAN: Enhancing the UAN to capture temporal relation among actions*. Department of Computing Science, University of Glasgow, February 1994, Department research report IS-94-02.
85. [Green 1986] Green M.W. A Survey of three Dialogue Models. *ACM Transactions on Graphics*, vol. 5, n° 3, 1986, pp. 244-275.
86. [Guittet 1995] Guittet L. *Contribution à l'Ingénierie des Interfaces Homme-Machine - Théorie des Interacteurs et Architecture H4 dans le système NODAOO*. Doctorat d'Université (PhD Thesis) : LISI/ENSMA, Université de Poitiers, 1995, 196p.
87. [Guittet, Girard, & Pierra 1997] Guittet L., Girard P., & Pierra G. Dialogue verification using the EXPRESS language. *Eurographics Workshop on Design, Specification and Verification of Interactive Systems (DSV-IS'97)*, Eds. M.D. Harrison & J.C. Torres, *Granada, Spain*, vol. Conference proceedings, 4-6 June 1997, pp. 457-471.
88. [Guittet & Pierra 1993] Guittet L. & Pierra G. Conception modulaire d'une application graphique interactive de conception technique : la notion d'interacteur. *Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'93)*, Pub. École Centrale Lyon, *Lyon*, Octobre 1993, pp. 151-156.
89. [Halbert 1984] Halbert D. *Programming by Example*. PhD Thesis : University of California, Berkeley, 1984, 121p.
90. [Hammouche 1995] Hammouche H. *De la modélisation des tâches utilisateurs à la spécification conceptuelle d'interfaces Homme-Machine*. Doctorat d'Université (PhD Thesis) : Université de Paris VI, Jussieu, 1995.
91. [Hanau & Lenorovitz 1980] Hanau P.R. & Lenorovitz D.R. Prototyping and Simulation Tools for user/Computer Dialogue Design. *Computer Graphics*, vol. 14, n° 3, 1980, .
92. [Harel 1988] Harel D. On Visual Formalisms. *Communications of the ACM*, vol. 31, n° 5, 1988, pp. 514-530.
93. [Hartson & Gray 1992] Hartson H.R. & Gray P. Temporal aspects of Tasks in the User Action Notation. *Human-Computer Interaction*, vol. 7, n° 1, 1992, pp. 1-45.
94. [Hashimoto & Myers 1992] Hashimoto O. & Myers B.A. Graphical Styles for Building User Interfaces by Demonstration. *ACM Symposium on User Interface Software and Technology (UIST'92)*, Pub. ACM/SIGCHI, *Monterey, California*, 15-18 November 1992, pp. 117-124.
95. [Hill 1986] Hill R.D. Supporting Concurrency, Communication and Synchronisation in Man-Computer Interaction : the Sassafras UIMS. *ACM Transaction on Graphic*, vol. 5, n° 3, 1986, pp. 179-210.
96. [Hill 1987] Hill R.D. Event Response Systems - A Technique for specifying Multi-Threaded Dialogs. *Human Factors in Computing Systems and Graphics Interface (CHI+GI'87)*, Eds. J.M. Caroll & P.P. Tanner, Pub. ACM Press, 1987, pp. 241-248.
97. [Hix & Hartson 1993] Hix D. & Hartson H.R. *Developping user interfaces: Ensuring usability through product & process*. Newyork, USA : John Wiley & Sons, inc., 1993.
98. [Hornby 1995] Hornby A.S. *Oxford Advanced Learner's Dictionary of Current English*. Oxford, UK : Oxford University Press, 1995, 1428p.
99. [Hussey & Carrington 1997] Hussey A. & Carrington D. *Specifying a Web Browser Interface Using Object-Z. in* Formal Methods for Human-Computer Interaction, Eds. P. Palanque & F. Paternò, Springer-Verlag, 1997, pp. 157-174.
100. [Hussey & Carrington 1998] Hussey A. & Carrington D. Which widgets? Deriving implementations from formal user-interface specifications. *Eurographics Workshop on Design, Specification and Validation of Interactive Systems (DSV-IS'98)*, Eds. P. Markopoulos & P. Johnson, Pub. SpringerWienNewYork, *Abingdon, UK*, 3-5 June 1998, pp. 206-224.

101. [Jackiw & Finzer 1993] Jackiw R.N. & Finzer W.F. *The Geometer's Sketchpad: Programming by Geometry*. *in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993, pp. 293-308.

102. [Jacob 1986] Jacob R.J.K. A Specification Language for Direct Manipulation User Interfaces. *ACM Transaction on Graphics*, vol. 5, n° 4, 1986, pp. 1203-1221.

103. [Jambon, Girard, & Boisdron 1999] Jambon F., Girard P., & Boisdron Y. Dialogue Validation from Task Analysis. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99), Universidade do Minho, Braga, Portugal*, vol. Conference proceedings, 2-4 June 1999, pp. 201-221.

104. [John & Kieras 1996] John B.E. & Kieras D.E. The GOMS Family of User Interface Analysis Techniques: Comparaison and Contrast. *ACM Transactions on Computer-Human Interaction*, vol. 3, n° 4, December 1996, pp. 320-351.

105. [Johnson & Johnson 1993] Johnson H. & Johnson P. ADEPT-advanced design environment for prototyping with task models. *Human Factors in Computing Systems (InterCHI'93)*, Pub. ACM Press, *Amsterdam, The Netherlands*, vol. adjunct Proceedings(2), 24-29 April 1993, pp. 56.

106. [Kolski 1997] Kolski C. *Interfaces Homme-Machine, Application aux systèmes industriels complexes*. Paris : Hermès, 1997, 477p.

107. [Kurlander 1993a] Kurlander D. *Chimera : Example-Based Graphical Editing*. *in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993a, pp. 271-292.

108. [Kurlander 1993b] Kurlander D. *Graphical Editing by Example*. PhD Thesis : Columbia University, 1993b.

109. [Kurlander & Feiner 1992] Kurlander D. & Feiner S. A History-Based Macro by Example System. *ACM Symposium on User Interface Software and Technology (UIST'92)*, Pub. ACM/SIGCHI, *Monterey, California*, 15-18 November 1992, pp. 99-115.

110. [Landay & Myers 1995] Landay J.A. & Myers B.A. Interactive Sketching for the Early Stages of User Interface Design. *Human Factors in Computing Systems (CHI'95)*, Pub. ACM/SIGCHI, *Denver, Colorado*, 7-11 May 1995, pp. 43-50.

111. [Lecolinet 1996] Lecolinet É. XXL: A Dual Approach for Building User Interfaces. *Symposium on User Interface Software and Technology (UIST'96)*, Pub. ACM/SIGCHI, *Seattle, USA*, November 1996, pp. 99-108.

112. [Lecolinet 1999] Lecolinet È. XXL: A Visual+Textual environment for Building Graphical User Interfaces. *Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)*, Eds. J. Vanderdonkt & A. Puerta, Pub. Kluwer Academic Publishers, *Louvain-la-Neuve, Belgique*, 21-23 October 1999, pp. 115-126.

113. [Lieberman 1982] Lieberman H. Constructing Graphical User Interfaces by Example. *Graphics Interface (GI'82), Toronto, Canada*, 17-21 May 1982, pp. 347-354.

114. [Lieberman 1990] Lieberman H. *Mémoire de synthèse*. Habilitation à Diriger les Recherches (HDR) : LITP, Paris VI, Jussieu, 1990, 49p.

115. [Lieberman 1993] Lieberman H. *Tinker: A Programming by Demonstration System for Beginning Programmers*. *in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993, pp. 49-66.

116. [Lieberman, Nardi, & Wright 1998] Lieberman H., Nardi B.A., & Wright D. Grammex : Defining Grammar by Example. *Human Factors in Computing Systems (CHI'98)*, Ed. ACM, Pub. ACM/SIGCHI, *Los Angeles, California*, vol. 2(2), 18-23 April 1998, pp. 11-12.

117. [Lim & Long 1994] Lim K.Y. & Long J. *The MUSE method for usability engineering*. United Kingdom : Cambridge University Press, 1994, 350p.

118. [Lonczewski & Schreiber 1996] Lonczewski F. & Schreiber S. The FUSE-System: an integrated User Interface Design Environment,. *Computer-Aided Design of User Interface (CADUI'96)*, Ed. J. Vanderdonckt, *Namur, Belgium*, 5-7 June 1996, pp. pp. 37-56.

119. [Loukipoudis 1996] Loukipoudis E.N. Object management in a programming-by-example, parametric, computer-aided-design system. *The Visual Computer*, vol. 6, 1996, pp. 296-306.

120. [Macdonald 1982] Macdonald A. Visual programming. *Datamation*, vol. 28, n° 11, 1982, pp. 132-140.

121. [Martin 1995] Martin B. *Contribution pour une nouvelle Approche du dialogue Homme-Machine en CFAO*. Doctorat d'Université (PhD Thesis) : Université de Metz, 1995, 188p.

122. [Maulsby 1993] Maulsby D. *The Turvy Experience: Simulating an Instructible Interface*. *in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993, pp. 239-270.

123. [Maulsby et al. 1992] Maulsby D., Witten I., Kittlitz K., & Fraceschun V. Inferring Graphical Procedures: the Compleat Metamouse. *Human-Computer Interaction*, vol. 7, n° 1, 1992, pp. 47-89.

124. [Maulsby, Witten, & Kittlitz 1989] Maulsby D.L., Witten I.H., & Kittlitz K.A. Metamouse : Specifying Graphical Procedures by Example. *SIGGRAPH'89, Boston*, 31 July - 4 August 1989, pp. 127-135.

125. [McDaniel 1999] McDaniel R.G. *Building Whole Applications Using Only Programming-by-Demonstration*. PhD Thesis : School Of Computer Science, Carnegie Mellon University, Pittsburg, 1999, 371p.

126. [McDaniel & Myers 1999] McDaniel R.G. & Myers B.A. Getting More Out Of Programming by Demonstration. *Human Factors in Computing Systems (CHI'99)*, Eds. M.G. Williams & M.W. Altom, Pub. ACM/SIGCHI, *Pittsburg*, 15-20 May 1999, pp. 442-449.

127. [Meinadier 1991] Meinadier J.P. *L'interface Utilisateur : pour une informatique plus conviviale*. Paris : Dunod Informatique, 1991, 221p.

128. [Moran 1981] Moran T.P. The Command language Grammar: A Representation for the User Interface of Interactive Systems. *International Journal of Man-Machine Studies*, vol. 15, 1981, pp. 3-50.

129. [Myers 1998] Myers A.B. Scripting Graphical Applications by Demonstration. *Human Factors in Computing Systems (CHI'98)*, Pub. ACM/SIGCHI, *Los Angeles, Californie*, 18-23 April 1998, pp. 534-541.

130. [Myers 1991] Myers B. Text Formatting by Demonstration. *Human Factors in Computing Systems (CHI'91)*, Pub. ACM/SIGCHI, *New Orleans, Louisiana*, 28 April - 2 May 1991, pp. 251-256.

131. [Myers 1986] Myers B.A. Visual Programming, Programming by Example, and Program Visualization : A Taxonomy. *Human Factors in Computing Systems (CHI'86)*, Pub. ACM/SIGCHI, *New-York*, 1986, pp. 59-66.

132. [Myers 1988] Myers B.A. *Creating User Interface by Demonstration*. Academic Press, 1988, 276p.

133. [Myers 1990a] Myers B.A. Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints. *ACM Transactions on Programming Languages and Systems*, vol. 12, n° 2, 1990a, pp. 143-177.

134. [Myers 1990b] Myers B.A. Invisible Programming. *IEEE Workshop on Visual Languages*, October 4-6 1990b, pp. 203-208.

135. [Myers 1990c] Myers B.A. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, vol. 1, n° 1, 1990c, pp. 97-123.

136. [Myers 1992a] *Languages for Developing User Interfaces*. Ed. Myers B.A., Boston, USA : Jones and Bartlett Publishers, 1992a, 456p.

137. [Myers 1992b] Myers B.A. *Report on the end-user programming working group. in* Languages for Developing User Interfaces, Ed. B.A. Myers, Boston : Jones & Bartlett, 1992b, pp. 343-366.

138. [Myers 1993a] Myers B.A. *PERIDOT: Creating User Interfaces by Demonstration. in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993a, pp. 125-154.

139. [Myers 1993b] Myers B.A. *Tourmaline: Text Formatting by Demonstration. in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993b, pp. 308-321.

140. [Myers 1993c] Myers B.A. *Why are User Interface Difficult to Design & Implement.* Carnegy Melon University, Juillet 1993 1993c, Rapport de Recherche CS-93-183.

141. [Myers 1995] Myers B.A. User Interface Software Tools. *ACM Transactions on Computer Human Interaction*, vol. 2, n° 1, 1995, pp. 64-103.

142. [Myers & Buxton 1986] Myers B.A. & Buxton W. Creating Highly-Interactive and Graphical User Interfaces by Demonstration. *SIGGRAPH'86*, Pub. ACM, *Dallas*, vol. 20 , n°4, August 18-22 1986, pp. 249-258.

143. [Myers et al. 1990] Myers B.A., Giuse D., Dannenberg R., Vander Zanden B., Kosbie D., Pervin E., Mickish A., & Marchal P. GARNET: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, vol. 23, n° 11, 1990, pp. 71-85.

144. [Myers, Hollan, & Cruz 1996] Myers B.A., Hollan J., & Cruz I. Strategic Directions in Human-Computer Interaction. *ACM Computing Surveys*, vol. 28, n° 4, 1996, pp. 794-809.

145. [Myers et al. 1997] Myers B.A., McDaniel R.G., Miller R.C., Ferrency A.S., Faulring A., Kyle B.D., Mickish A., Klimovitski A., & Doane P. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering,*, vol. 23, n° 6, 1997, pp. 347-365.

146. [Nardi 1993] Nardi B.A. *A Small Matter of Programming, Perspectives on End User Computing*. Cambridge, Massachusetts : The MIT Press, 1993, 157p.

147. [Nelson 1985] Nelson G. Juno, a constraint-based graphics system. *Computer Gaphics*, vol. 19, n° 3, 1985, pp. 235-243.

148. [Newell, Parden, & Parden 1983] Newell R., Parden G., & Parden P. Parametric Design in MEDUSA System. *CAPE'83, Amsterdam*, Apr. 25-28 1983.

149. [Newman 1979] Newman. *Principes of Interactive Computer Graphics*. New-York : Mc Graw Hill, 1979.

150. [Nix 1985] Nix R.P. Editing by example. *ACM Transactions on Programming Languages and Systems*, vol. 7, n° 4, 1985, pp. 600-621.

151. [Norman 1986] Norman D. *User Centered System Design*. Lawrence Erlbaum Associates, 1986.

152. [Olsen 1983] Olsen D.R. SYNGRAPH : a Graphical User Interface Generator. *ACM Transaction on Graphics*, vol. 23, n° 3, 1983, pp. 43-50.

153. [Olsen 1992] Olsen D.R. *User Interface Management Systems: Models and Algorithms*. San Mateo (CA), USA : Morgan Kaufmann Publisher, 1992, 231p.

154. [Olsen 1998] Olsen D.R. *Developping User Interfaces*. San Francisco, Californie : Morgan Kaufmann Publishers, 1998, 414p.

155. [Olsen, Ahlstrom, & Kohlert 1995] Olsen D.R., Ahlstrom B., & Kohlert D. Building Geometry-based Widgets by Example. *Human Factors in Computing Systems (CHI'95)*, Pub. ACM/SIGCHI, *Denver, Colorado*, 7-11 May 1995, pp. 35-42.

156. [Olsen & Dance 1988] Olsen D.R. & Dance J.R. Macros by Example in a Graphical UIMS. *IEEE Computer Graphics and Applications*, vol. 12, n° 1, 1988, pp. 68-78.

157. [Olsen 1986] Olsen R. Mike : the Menu Interaction Kontrol Environment. *ACM Transaction on Graphics*, vol. 5, n° 3, 1986, pp. 318-344.

158. [Owen 1991] Owen J. Algebraic Solution for Geometry from Dimensional Constraints. *ACM Symposium on Foundations of Solid Modeling*, Pub. ACM/SIGGRAPH, *Austin, Texas*, 8-10 May 1991, pp. 397-407.

159. [Oxford 1983] Oxford. *Dictionary of Computing*. Oxford : Oxford University Press, 1983.

160. [Palanque 1992] Palanque P. *Modélisation par Objets Coopératifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur*. Doctorat d'Université (PhD Thesis) : LIS, Université de Toulouse I, Toulouse, 1992, 320p.

161. [Palanque 1997] Palanque P. *Spécifications formelles et systèmes interactifs : vers des systèmes fiables et utilisables*. Habilitation à Diriger les Recherches (HDR) : LIS/FROGIS, Université de Toulouse I, Toulouse, 1997, 187p.

162. [Palanque & Bastide 1994] Palanque P. & Bastide R. Petri-Net Based Design of User Interfaces using the Interactive Cooperative Objects Formalism. *Interactive Systems: Design, Specification, and Verification (DSV-IS'94)*, Ed. F. Paternò, Pub. Springer, *Bocca di Magra, Italy*, 8-10 June 1994, pp. 383-400.

163. [Palanque & Bastide 1995] Palanque P. & Bastide R. *Task Models - System Models: a Formal Bridge over the Gap*. *in* Critical Issues in User Interface Engineering, Eds. P. Palanque & D. Benyon, London : Springer-Verlag, 1995, pp. 65-80.

164. [Palanque et al. 1996] Palanque P., Paternò F., Bastide R., & Mezzanotte M. Toward an Integrated Proposal for Interactive Systems design based on TLIM and MICO. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96)*, Eds. F. Bodart & J. Vanderdonckt, Ser. SpringerComputerScience, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Namur, Belgium*, 5-7 June 1996, pp. 162-187.

165. [Paternò 1994] Paternò F. A Theory of User-Interaction Objects. *Journal of Visual Languages and Computing*, vol. 5, n° 3, 1994, pp. 227-249.

166. [Paternò & Faconti 1994] Paternò F. & Faconti G.P. A semantics-based approach for the design and implementation of interaction objects. *Computer Graphics Forum*, vol. 13, n° 3, 1994, pp. 195-204.

167. [Paternò, Mancini, & Meniconi 1997] Paternò F., Mancini C., & Meniconi S. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. *IFIP TC13 human-computer interaction conference (INTERACT'97), Sydney, Australia*, 1997, pp. 362-369.

168. [Paternò & Mezzanotte 1995] Paternò F. & Mezzanotte M. Formal verification of undesired behaviours in the CERD case study. *IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI'95)*, Eds. L.J. Bass & C. Unger, Pub. Chapman & Hall, *Grand Targhee Resort (Yellowstone Park), USA*, 14-18 August 1995, pp. 213-226.

169. [Patry 1996a] Patry G. Vers la Génération Automatique d'Applications Interactives. *Colloque d'Automatique et de Génie Informatique (AGI'96), Tours*, 6-7 Juin 1996a, pp. 251-254.

170. [Patry 1996b] Patry G. Vers la Génération Interactive d'Applications Graphiques. *Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'96)*, Pub. Cépaduès, *Grenoble*, 16-18 Septembre 1996 1996b, pp. 149-150.

171. [Patry 1998] Patry G. *Étude de différents modèles d'architecture logicielle*. LISI/ENSMA, Décembre 1998, Rapport de Recherche.

172. [Patry 1999a] Patry G. *Contribution à la conception du dialogue Homme Machine dans les applications graphiques interactives de conception technique : le système GIPSE*. Doctorat d'Université (PhD Thesis) : LISI/ENSMA, Université de Poitiers, 1999a, 199p.

173. [Patry 1999b] Patry G. Évaluation des dialogues structurés. *Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'99)*, Eds. J. Nanard & P. Girard, Pub. Cépaduès, *Montpellier*, 22-26 Novembre 1999b, pp. en cours.

174. [Patry & Girard 1997a] Patry G. & Girard P. From Adaptable Interfaces to Model-Based Interface Development: The GIPSE Project. *ERCIM Workshop on User Interfaces for All (UI4ALL'97)*, Ed. N. Carbonell, Pub. INRIA Lorraine, *Obernai, France*, 3-4 november 1997a, pp. 127-133.

175. [Patry & Girard 1997b] Patry G. & Girard P. Techniques d'interaction : intégrer simplement l'exploration dans les dialogues structurés. *Colloque de l'AFIG'97, Rennes*, 3-5 Décembre 1997b, pp. 1-10.

176. [Patry & Girard 1998] Patry G. & Girard P. Ergonomie des dialogues structurés : Amélioration de l'évaluation de la tâche courante. *Ergonomie et Informatique Avancée (Ergo'IA 98), Biarritz, France*, 4-6 Octobre 1998, pp. 333-335.

177. [Patry & Girard 1999] Patry G. & Girard P. GIPSE: a Model-Based System for CAD. *Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)*, Eds. J. Vanderdonkt & A. Puerta, Pub. Kluwer Academic Publishers, *Louvain-la-Neuve, Belgique*, 21-23 October 1999, pp. 61-72.

178. [Piernot & Yvon 1993] Piernot P.P. & Yvon M.P. *The AIDE Project: An Application-Independent Demonstrational Environment. in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993, pp. 383-402.

179. [Pierra 1991] Pierra. *Les bases de la programmation et du Génie Logiciel*. Paris : Dunod informatique, 1991, 653p.

180. [Pierra 1995] Pierra G. Towards a taxonomy for interactive graphics systems. *Eurographics Workshop on Design, Specification, Verification of Interactive Systems*, Eds. P. Palanque & R. Bastide, Ser. Springer Computer Science, Ed. Ser. Eurographics, Pub. Springer-Verlag, *Bonas*, 7-9 June 1995, pp. 362-370.

181. [Pierra & Aït Ameur 1994] Pierra G. & Aït Ameur Y. *Logical Model for Parts Libraries*. ISO-CD 13584-20, 1994.

182. [Pierra et al. 1996] Pierra G., Aït Ameur Y., Besnard F., Girard P., & Potier J.-C. A General Framework for Parametric Product Model within STEP and Parts Library. *European PDT Days*, Pub. PDTAG-AM, *London, UK*, 18-19 April 1996, pp. 69-104.

183. [Pierra, Girard, & Guittet 1995] Pierra G., Girard P., & Guittet L. Towards precise architecture models for computer graphics: the H4 architecture. *Eurographics Workshop on Design, Specification, Verification of Interactive Systems*, Eds. P. Palanque & R. Bastide, *Bonas*, vol. Conference proceedings, 7-9 June 1995.

184. [Pierra, Potier, & Girard 1994] Pierra G., Potier J.-C., & Girard P. Design and Exchange of Parametric Models for Parts Library. *27th International Symposium on Advanced Transportation Applications, ISATA'94, Aachen, Germany*, 31st October - 4 November 1994, pp. 397-404.

185. [Pierra, Potier, & Girard 1996] Pierra G., Potier J.-C., & Girard P. *The EBP system : Example Based Programming for Parametric Design*. *in* Modelling and Graphics in Science and Technology, Eds. J. Teixeira & J. Rix, Springer-Verlag, 1996, pp. 124-140.

186. [Potier 1995] Potier J.-C. *Conception sur exemple, mise au point et génération de programmes portables de géométrie paramétrée dans le système EBP*. Doctorat d'Université (PhD Thesis) : LISI/ENSMA, Université de Poitiers, 1995, 140p.

187. [Potier & Girard 1993] Potier J.-C. & Girard P. La programmation interactive graphique sur exemple. *Colloque d'Automatique et Génie Informatique (AGI'93), La Rochelle, France*, Mai 1993, pp. 132-135.

188. [Potier, Girard, & Pierra 1993] Potier J.-C., Girard P., & Pierra G. Nouvelles Interfaces Homme-Machine pour la Programmation : La Programmation Interactive Graphique sur Exemple. *Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'93)*, Pub. École Centrale Lyon, *Lyon*, 19-20 Octobre 1993, pp. 139-144.

189. [Potier et al. 1995] Potier J.-C., Girard P., Pierra G., & Besnard F. Génération graphique interactive de programmes de géométrie paramétrée. *Revue d'Automatique et de Productique Appliquée (RAPA)*, vol. 8, n° 2-3, 1995, pp. 229-234.

190. [Potier et al. 1997] Potier J.-C., Girard P., Pierra G., Wilman N., Mahir A., & Elu P. Intelligent electronic catalogue generation. *International Symposium on Glogal Engineering Networking (GEN'97), Antwerp, Belgium*, 22-24 April 1997, pp. 353-367.

191. [Preece et al. 1994] Preece J., Rogers Y., Sharp H., Benyon D., Holland S., & Carey T. *Human-Computer Interaction*. Wokingham, England : Addison Wesley Publishing Company, 1994, 773p.

192. [Puerta 1996] Puerta A. The MECANO project : comprehensive and integrated support for Model-Based Interface development. *Computer-Aided Design of User Interface (CADUI'96)*, Ed. J. Vanderdonckt, Pub. Presse Universitaire de Namur, *Namur, Belgium*, 5-7 June 1996, pp. 19-35.

193. [Puerta & Eisenstein 1998] Puerta A. & Eisenstein J. Interactively Mapping Task Model to Interfaces in Mobi-D. *Eurographics Workshop on Design, Specification and Validation of Interactive Systems (DSV-IS'98)*, Eds. P. Markopoulos & P. Johnson, *Abingdon, UK*, vol. Proceedings, 3-5 June 1998, pp. 261-274.

194. [Puerta 1997] Puerta A.R. A Model-Based Interface Development Environment. *IEEE Software*, vol. 14, n° 4, 1997, pp. 40-47.

195. [Qiang et al. 1997] Qiang L., Wei L., Ke X., & Jiaguang S. An Event-Driven and Object Oriented FrameWork for Human Computer Interface of CAD System. *CAD & Graphics'97*, Pub. International Academic Publishers, *Shenzen, China*, vol. 1(2), 2-5 Dec. 1997, pp. 42-45.

196. [Raeder 1985] Raeder G. A Survey of Current Graphical Programming Techniques. *IEEE Computer*, vol. 18, n° 8, 1985, pp. 11-25.

197. [Robert 1971] Robert P. *Dictionnaire alphabétique et analogique de la langue française*. Ed. Le Robert, 1971, 900p.

198. [Roller 1990] Roller D. *Dimension-Driven Geometry in CAD: a Survey*. *in* Theory and Practice of Geometric Modeling, , Springer-Verlag, 1990, pp. 509-523.

199. [Rumbaugh et al. 1991] Rumbaugh J., Blaha M., Premerlani W., Eddy F., & Lorensen W. *Object-Oriented Modeling and Design*. New York : Prentice-Hall International Editions, 1991, 500p.

200. [Ryan, Fiadeiro, & Maibaum 1991] Ryan M., Fiadeiro J., & Maibaum T. *Sharing actions and attributes in modal action logic*. *in* Theoretical Aspects of Computer Science, Eds. T. Ito & A.R. Meyer, Springer-Verlag, 1991, pp. 569-593.

201. [Sassin 1994] Sassin M. Creating User-Intended Programs with Programming by Demonstration. *IEEE Symposium on Visual Languages*, Eds. A.L. Ambler & T.D. Kimura, Pub. IEEE, *Saint Louis, Missouri*, 4-7 October 1994, pp. 153-160.

202. [Scapin & Pierret-Golbreich 1989] Scapin D.L. & Pierret-Golbreich C. MAD : Une méthode analytique de description des tâches. *Colloque sur l'Ingénierie des Interfaces Homme-Machine (IHM'89), Sophia-Antipolis, France*, Mai 1989, pp. 131-148.

203. [Scapin & Pierret-Golbreich 1990] Scapin D.L. & Pierret-Golbreich C. *Towards a method for task description : MAD*. *in* Working with display units, Eds. L. Berliguet & D. Berthelette, Elsevier Science Publishers, North-Holland, 1990, pp. 371-380.

204. [Schiele & Green 1990] Schiele F. & Green T. *HCI formalisms and cognitive psychology: the case of Task-Acion Grammar*. *in* Formal Methods in Human-Computer Interaction, Eds. M.D. Harrison & H.W. Thimbleby, Cambridge : Cambridge University Press, 1990, pp. 9-62.

205. [Shah & Mäntylä 1995] Shah J.J. & Mäntylä M. *Parametric and Feature-based CAD/CAM: Concepts, Techniques and Applications*. New York : John Wiley & Sons, 1995, 619p.

206. [Shaw 1975] Shaw D.E. Inferring Lisp Programs from Examples. *4th International Joint Conference on Artificial Intelligence, TBILISI, USSR*, September 1975, pp. 260-267.

207. [Shepherd 1989] Shepherd A. *Analysis and training in information technology tasks*. *in* Task Analysis for Human-Computer Interaction, Ed. D. Diaper, Chichester, USA : Ellis Horwood, 1989, pp. 15-55.

208. [Shneiderman 1983] Shneiderman B. Direct Manipulation: a Step beyond Programming Languages. *IEEE Computer*, vol. 16, n° 8, 1983, pp. 57-69.

209. [Shneiderman 1998] Shneiderman B. *Designing the User Interface*. Addison-Wesley, 1998, 639p.

210. [Shu 1986] Shu N. *Visual Programming Languages: a perspective and a dimensional analysis*. *in* Visual Languages, Eds. S.-K. Chang, T. Ichikawa & P. Ligomenides, New-York : Plenum Press, 1986, pp. 10-34.

211. [Shumerck 1986] Shumerck K. MacApp : An application Framework. *Byte*, vol. 11, n° 8, 1986, pp. 189-193.

212. [Singh & Green 1991] Singh G. & Green M. Automating the Lexical and Syntaxic Design of Graphical User Interfaces : the UoA*UIMS. *ACM Transaction on Graphics*, vol. 10, n° 3, 1991, pp. 213-254.

213. [Smith & Cypher 1995] Smith D. & Cypher A. KidSim : Child Constructible simulation. *Imagina'95, Monte-Carlo, Février*, 1995, pp. 87-99.

214. [Solano & Brunet 1994] Solano L. & Brunet P. Constructive Constraint-Based Model for Parametric CAD Systems. *Computer Aided Design*, vol. 26, n° 8, 1994, pp. 614-621.

215. [Spivey 1988] Spivey J.M. *The Z notation: A Reference Manual*. Prentice Hall Int., 1988.

216. [Stastko 1991] Stastko J. Using Direct Manipulation to Build Algorithms Animation by Demonstration. *Human Factors in Computing Systems (CHI'91)*, Pub. ACM/SIGCHI, *New Orleans, Louisiana*, 27 April - 2 May 1991, pp. 307-314.

217. [Staub & Maier 1992] Staub G. & Maier M. *ECCO Toolkit: an Environment for the Evaluation of EXPRESS Models and the Development of STEP based IT Applications*, Universitat of Karsruhe, 1992.

218. [Steria Méditerranée 1997] Steria Méditerranée. *Atelier B*, Ed. 3.0, 1997.

219. [Szekely 1996] Szekely P. Retrospective and challenge for Model Based Interface Development. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96)*, Eds. F. Bodart & J. Vanderdonckt, Ser. SpringerComputerScience, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Namur, Belgium*, 5-7 June 1996, pp. 1-27.

220. [Szekely, Luo, & Neches 1993] Szekely P., Luo P., & Neches R. Beyond Interface Builders: Model-Based Interface Tools. *InterCHI93*, 1993, pp. 383-390.

221. [Szekely et al. 1995] Szekely P., Sukaviriya P., Castells P., Muthukumarasamy J., & E. Salcher. Declarative interface models for user interface construction tools : the MASTERMIND approach. *IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI'95)*, Eds. L.J. Bass & C. Unger, Pub. Chapman & Hall, *Grand Targhee Resort (Yellowstone Park), USA*, 14-18 August 1995, pp. 120-150.

222. [Tam, Maulsby, & Puerta 1998] Tam C.-M.R., Maulsby D., & Puerta A.R. U-TEL: A Tool for Eliciting User Task Models from Domaine Experts. *IUI98: International Conference on Intelligent user Interfaces, San Francisco, USA*, January 1998, pp. 77-80.

223. [Tarby 1993] Tarby J.-C. *Gestion Automatique de Dialogue Homme-Machine à partir de spécification conceptuelles*. Doctorat d'Université (PhD Thesis) : LIS, Université de Toulouse I, Toulouse, 1993, 272p.

224. [Texier & Guittet 1999] Texier G. & Guittet L. User defined objects are first class citizens. *Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)*, Eds. J. Vanderdonkt & A. Puerta, Pub. Kluwer Academic Publishers, *Louvain-la-Neuve, Belgique*, 21-23 October 1999, pp. 231-244.

225. [Torres et al. 1996] Torres J.C., Gea M., Gutierrez F.L., Cabrera M., & Rodriguez M. GRALPLA: an Algebraic Specification Language for Interactive Graphic Systems. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96)*, Eds. F. Bodart & J. Vanderdonckt, Ser. SpringerComputerScience, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Namur, Belgium*, 5-7 June 1996, pp. 272-291.

226. [van den Bos & Laffra 1990] van den Bos J. & Laffra C. Project DIGIS : Building Interactive Application by Direct Manipulation. *Computer Graphic Forum*, vol. 9, 1990, pp. 181-193.

227. [Van Emmerick 1989a] Van Emmerick M. ASM, a modeller for geometric and topologic parametrization. *Internationnal conférence on Computer-Aided Design & Computer Graphics*, 1989a, pp. 166-171.

228. [Van Emmerick 1989b] Van Emmerick M. A system for graphical interaction on parametrized models. *Eurographics, Amsterdam*, 1989b, pp. 233-242.

229. [Van Emmerick 1991] Van Emmerick M. *Interactive Design of Parametrized 3D models by Direct Manipulation*. PhD Thesis : Université de Delft, Delft, Netherland, 1991, 141p.

230. [van Weilie, van der Veer, & Eliëns 1999] van Weilie M., van der Veer G.C., & Eliëns A. Usability Properties in Dialog Models. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99), Universidade do Minho, Braga, Portugal*, vol. Conference proceedings, 2-4 June 1999, pp. 239-254.

231. [Vanderdonckt 1999] Vanderdonckt J. Computer-Aided Design of Menu Bar and Pull-Down Menus for Business Oriented Applications. *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'99), Universidade do Minho, Braga, Portugal*, vol. Conference proceedings, 2-4 June 1999, pp. 73-88.

232. [Verroust 1990] Verroust A. Construction d'objets géométriques définis par des contraintes. *Bigre*, vol. 67, n° 1, 1990, pp. 62-74.

233. [Wallace & Anderson 1993] Wallace M.D. & Anderson T.J. Approach to Interface Design. *Interacting with Computer*, vol. 5, n° 3, 1993, pp. 259-278.

234. [Wiecha, Bennet, & al 1989] Wiecha C., Bennet W., & al e. Generating Higly Interactive User Interfaces. *Human Factors in Computing Systems (CHI'89)*, Eds. K. Bice & C. Lewis, Pub. ACM/SIGCHI, *Austin, USA*, 30 April-4 May 1989 1989, pp. 277-282.

235. [Wiecha et al. 1990] Wiecha C., Bennet W., Boies S., Gould J., & Greene S. ITS: a tool for rapidly developing interactive applications. *ACM Transactions on Information Systems*, vol. 8, n° 3, 1990, pp. 204-236.

236. [Wilde 1993] Wilde N. WYSIWIC (What You See Is What You Compute) Spreadsheet. *IEEE Symposium on Visual Languages*, Pub. IEEE, *Bergen, Norway*, Aug. 24-27 1993, pp. 72-76.

237. [Witten & Mo 1993] Witten I.H. & Mo D. *TELS: Learning Text Editing Tasks from Examples*. *in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993, pp. 183-204.

238. [Wolber 1996] Wolber D. Pavlov : Programming by Stimulus-Response Demonstration. *Human Factors in Computing Systems (CHI'96)*, Ed. M. Tauber, Pub. ACM/SIGCHI, *Vancouver, Canada*, 13-18 April 1996, pp. 252-269.

239. [Yoshimoto et al. 1986] Yoshimoto I., Monden N., Hirakawa M., Tanaka M., & Ichikawa M. Interactive Iconic Programming Facility in HI-VISUAL. *IEEE Workshop on Visual Languages*, Pub. IEEE, *Dallas, Texas*, 25-27 june 1986, pp. 34-41.