Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Ecole Doctorale des Sciences Pour l'Ingénieur ▪ Université de Poitiers

THESE

pour l'obtention du grade de
DOCTEUR DE L'UNIVERSITE DE POITIERS
(Faculté des Sciences Fondamentales et Appliquées)
(Diplôme National — Arrêté du 30 mars 1992)

Secteur de Recherche : INFORMATIQUE

Présentée et soutenue publiquement
devant la Commission d'Examen le 29 Novembre 2002 par

Fabrice DEPAULIS

Vers un environnement générique d'aide au développement d'applications interactives de simulations de métamorphoses

Directeur de Thèse : ***Patrick GIRARD***

JURY

Rapporteurs :	Laurence NIGAY	Maître de Conférence, Université J. Fournier, Grenoble
	Henry LIEBERMAN	Research Scientist, MIT Media Laboratory, USA
Examinateurs :	Patrick GIRARD	Professeur, Université de Poitiers, Poitiers
	Pascal LIENHARDT	Professeur, Université de Poitiers, Poitiers
	Guy PIERRA	Professeur, ENSMA, Poitiers

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Laboratoire d'Informatique Scientifique et Industrielle (E.A. 1232)
Téléport 2 - 1, avenue Clément Ader - BP 40109 - 86961 Futuroscope Chasseneuil cedex - France
✆ (+33/0) 5 49 49 80 63 - (+33/0) 5 49 49 80 64 - www.lisi.ensma.fr

Table des matières

Introduction générale

« The much deeper result is not only the visual beauty of nature but insights into the way nature works. »

Przemyslaw PRUSINKIEWICZ

Depuis maintenant de nombreuses années, la production d'images de synthèse et d'animations par ordinateur connaît un intérêt remarquable. Celles-ci sont de plus en plus présentes dans la vie de tous les jours, que ce soit dans les jeux vidéos, les effets spéciaux au cinéma, les films d'animation, les publicités ou même l'art contemporain. Mais les applications moins ludiques ne sont pas en reste et les animations sont également utilisées pour simuler des phénomènes physiques comme l'influence de la gravité et de la masse sur des objets en mouvement, la déformation de matériaux subissant des collisions ou l'articulation de solides soumis à un certain nombre de forces extérieures. Elles sont également utilisées pour étudier l'évolution dans le temps d'objets naturels, comme par exemple la croissance de végétaux.

Pour construire de telles animations, de nombreuses méthodes ont été développées, chacune d'entre elle correspondant à un contexte bien particulier. Certaines reposent sur l'application de lois physiques, d'autres sur celle de lois issues de la botanique, d'autres encore sur l'étude et le recourt à des comportements, etc. On distingue aussi de nombreux systèmes d'animation selon qu'ils sont interactifs, basés sur des techniques issues de l'intelligence artificielle ou qu'ils associent simplement des scripts aux différents objets à animer [Terraz 1994].

Parmi toutes ces applications et toutes ces méthodes, on peut porter une attention particulière aux simulations de phénomènes naturels. Celles-ci peuvent être servies par de nombreuses techniques (fractales par exemple) produisant des résultats spectaculaires, mais seules les méthodes à base topologique permettent de s'intéresser aux transformations structurelles apportées aux objets. Elles consistent à décomposer un objet en un ensemble de sous-objets et à associer des comportements à chacun d'entre eux. La simulation globale revient alors à parcourir l'objet principal en appliquant, pour chaque élément, le scénario qui lui est attaché.

Malheureusement, le contrôle de ce que l'on appelle les métamorphoses se fait exclusivement de manière algorithmique. La conséquence directe de cet état de fait est que, alors que la majorité des autres méthodes, qui permettent de modifier les propriétés géométriques, trouvent leur application au sein d'environnements et de systèmes interactifs « grand public », comme 3D Studio™, les méthodes à base topologique restent réservées à une poignées de programmeurs, seuls capables de manipuler les concepts que nécessite leur mise en pratique. Cette approche pose au moins deux problèmes majeurs. Tout d'abord, il y a un décalage permanent entre la représentation textuelle du programme et son exécution graphique. D'autre part, dans un domaine pour lequel la notion de prototypage est aussi déterminante, il semblerait naturel que l'on puisse passer instantanément du mode de conception au mode d'exécution sans avoir à projeter mentalement une représentation à l'intérieur d'une autre.

L'objectif de nos travaux est de montrer qu'il est possible, non seulement de créer une application interactive capable de générer les programmes correspondant à ces métamorphoses, mais encore d'établir un environnement d'aide au développement de telles applications. Ainsi, nous voulons que le concepteur d'un système de modélisation de métamorphoses soit en mesure, à moindre frais, de greffer un modèle existant sur un cadre d'application. Le système ainsi conçu bénéficiera d'un moteur d'enregistrement et de rejeu offrant à un utilisateur la possibilité de créer des programmes à partir des transformations exécutées interactivement.

Dans le premier chapitre de ce manuscrit, nous étudions les différentes approches permettant de modéliser des animations. En décomposant cette étude selon les systèmes de contrôles et les modèles d'animation, nous balayons le spectre des méthodes existantes. Cette analyse systématique nous permet de voir que seules les méthodes basées sur la topologie permettent de représenter les modifications structurelles d'un objet, et que l'unique solution pour contrôler ces transformations consiste à utiliser un algorithme. Forts de cette constatation, nous terminons le chapitre par une étude de cas permettant de mettre en évidence les concepts de programmation intervenant dans la description d'une métamorphose.

La seconde partie est consacrée à l'étude des outils de conception permettant de mettre en place une application de création de métamorphoses. En classant les systèmes de ce type dans le domaine des Applications Graphiques de Conception Technique, nous mettons en évidence le besoin d'utiliser un dialogue structuré, autorisant la décomposition des tâches en un ensemble de buts/sous-buts plus faciles à atteindre pour l'utilisateur. A partir de ce postulat, nous étudions plusieurs modèles d'architecture par rapport à la description de ce dialogue particulier et nous montrons en quoi l'architecture H^4 répond mieux que les autres à nos besoins.

Dans un deuxième temps, nous passons en revue les différentes approches existantes en matière de programmation interactive à travers une classification des systèmes de programmation. Cette analyse nous permet de montrer pourquoi la Programmation sur Exemple (PsE) semble la méthode la plus adaptée à la création de programmes tels que

ceux permettant la description de métamorphoses. Enfin, nous approfondissons les concepts et les difficultés posés par l'approche choisie en examinant les solutions proposées dans la littérature.

Dans le dernier chapitre, nous décrivons d'abord successivement les solutions interactives que nous avons apportées à H4. En effet, le dialogue préfixé sur lequel repose le dialogue d'une application modélisée avec cette architecture n'est pas satisfaisant dans un certain nombre de phases de dialogues, que l'on voudrait plus conviviales. Il n'est en particulier pas possible de modéliser un dialogue par Manipulation Directe. Nous montrons comment modifier légèrement le Contrôleur de Dialogue pour rendre la cohabitation entre Manipulation Directe et Dialogue Structuré possible.

Nous nous intéressons ensuite à la possibilité de rendre la programmation du Contrôleur de Dialogue de H4 plus aisée : en intégrant un éditeur graphique et interactif, nous montrons comment automatiser la génération d'une grande partie de son code, et comment vérifier certaines propriétés du dialogue, éventuellement sources d'erreur.

La dernière section est consacrée à l'étude de l'environnement d'aide à la conception d'applications de métamorphoses. En nous basant sur les propriétés communes des modèles à base topologiques, nous avons mis sur pied un modèle générique sur lequel s'appuie un moteur de rejeu de scénario. Celui-ci, utilisant le fonctionnement du Contrôleur de Dialogue de H4, est capable d'interpréter des scénarii basés sur ce modèle. D'autre part, nous reprenons les principes d'enregistrement interactif et de généralisation issus de la programmation sur exemple pour montrer comment les appliquer au contexte qui nous intéresse.

Enfin, nous décrivons succinctement une application dont le modèle a été greffé sur le modèle générique, et qui bénéficie de ce fait des apports du moteur de rejeu. Elle implémente de surcroît quelques uns des principes d'enregistrement de programmes énoncés précédemment, et permet ainsi à un utilisateur de décrire des métamorphoses d'arbres filaires et de les rejouer en en modifiant certains paramètres.

Chapitre 1

Simulation d'animations et de métamorphoses

Résumé. *La production d'images et d'animations de « synthèse », c'est-à-dire par ordinateur, est depuis de nombreuses années un domaine très attractif de l'informatique. La raison principale en est la grande variété de domaines d'applications, qui vont du jeu vidéo à l'industrie du cinéma en passant par l'art contemporain.*
Cet intérêt a motivé de nombreux travaux et autant de techniques différentes pour permettre à un concepteur de représenter animations et déformations : certaines prennent en compte les propriétés physiques de l'objets, d'autres ne s'intéressent qu'à des déformations plus ou moins superficielles. Parmi toutes ces méthodes, celles dites « à base topologique » présentent la particularité de s'intéresser plus précisément aux modifications structurelles des objets. Elles sont le fer de lance de la modélisation d'animations d'objets naturels et reposent sur un certain nombre de principes communs : structuration des objets en sous-objets, discrétisation du temps, association de comportements à chaque sous-objet. Le principal défaut de cette approche repose sur le contrôle de l'animation, intrinsèquement algorithmique, qui empêche de manière rédhibitoire l'utilisation d'une méthode conviviale.
Dans l'objectif de palier à ce manque grâce à des procédés issus de la programmation interactive, nous présentons le domaine de l'animation en définissant quelques-uns des termes les plus utilisés et en exposons les techniques les plus répandues. Nous nous intéressons ensuite plus particulièrement aux méthodes basées sur la topologie et nous nous livrons à une étude d'exemples pour comprendre les besoins nécessaires à leur manipulation.

1 Introduction

La production d'images et d'animations par ordinateur connaît depuis maintenant de nombreuses années un engouement remarquable, justifié en grande partie par les nombreuses applications de ce domaine.

Dans l'industrie d'abord, au delà du dessin de Conception Assistée par Ordinateur (CAO) destiné à la création d'une pièce mécanique, la production d'animations par ordinateur est utilisée à des fins aussi diverses que le contrôle de systèmes, la simulation d'engins avec retour visuel ou la robotique.

Dans le domaine culturel, les images numériques trouvent aussi leur place : il est possible de numériser des objets de musées, des tableaux, des sculptures, des éléments de sites archéologiques et des éléments architecturaux dans le but d'obtenir des images numériques 3D à haute résolution de qualité archives. Les enregistrements numériques 3D peuvent ensuite servir pour une multitude d'applications de recherche et de conservation, ainsi qu'à des fins archéologiques et architecturales. Ces données permettent aussi de fabriquer des répliques, de monter des expositions interactives et des représentations virtuelles 3D ([Corcoran, et al. 2002]). Les images de synthèse constituent en outre une nouvelle technique d'expression artistique. Des formes et des couleurs nouvelles associées au mouvement permettent à l'artiste des créations originales. Plusieurs festivals, comme « Imagina » en France, témoignent de l'existence d'un public intéressé par ce nouveau développement de l'activité artistique.

Dans le domaine scientifique ou pédagogique, les animations peuvent être utilisées pour illustrer des phénomènes ou effectuer des simulations de comportements.

Enfin, les industries de l'audiovisuel et du jeu vidéo sont évidemment, par nature, de grands consommateurs d'images et de séquences d'animation. Les techniques de modélisation d'animation permettent de créer des effets spéciaux en 3D numérique, intégrés ou non aux films cinématographiques, publicitaires, génériques et aux clips vidéo. D'autre part, les images et les animations en trois dimensions sont de plus en plus utilisées dans les jeux vidéos pour créer des mondes virtuels chaque jour un peu plus réalistes.

[Hachette 2002] donne les définitions suivantes :

> ***Animer*** *v. trans. 1. Donner vie (à). Dieu anime toute créature. Par ext. La lune animait le paysage. Sculpteur qui anime la matière.*
>
> ***Animation*** *n. fém. 1. Caractère de ce qui prend vie, de ce qui présente une activité, du mouvement. L'animation d'une foire.*

Animer signifie donc apporter la vie. Néanmoins, le principal objectif de l'animation est de synthétiser des effets désirés évoluant au cours du temps : le mouvement (changement de position et/ou d'orientation), la transformation d'éléments non géométriques (couleur, texture, etc.) ainsi que les paramètres de l'observateur.

L'animation englobe aussi bien la simulation du monde réel par des phénomènes physiques (comme l'articulation de corps rigides sur lesquels on applique des forces) ou par l'évolution d'objets naturels (comme la croissance de plante), que la production de séquences d'images.

Pour modéliser ces animations, de nombreuses méthodes différentes ont été mises au point :

- Méthodes basées sur des lois physiques ;
- Méthodes basées sur des connaissances botaniques ;
- Méthodes basées sur des connaissances du comportement ;
- Méthodes basées sur une description directe et géométrique d'un mouvement.

D'autre part, les systèmes utilisant ces méthodes peuvent être interactifs, utiliser une association entre des scripts et des objets ou encore faire appel à des techniques issues de l'intelligence artificielle.

En marge de ces différentes approches, les méthodes dites « à base topologique » poursuivent un but très particulier : permettre la simulation de l'évolution d'êtres vivants ou de phénomènes naturels. Ces méthodes ont pour principe commun de générer une séquence d'objets (O_1, …, O_n), O_t étant l'objet à l'instant t, t étant un temps discret. Cette séquence est appelée **métamorphose**.

Les applications de telles simulations sont multiples : il peut s'agir pour un biologiste d'étudier théoriquement les développements de plantes naturelles, pour un créateur de mondes virtuels de représenter le plus fidèlement possible la forme correcte d'un végétal (Figure 1), ou pour un architecte de simuler l'état d'un lieu et de sa flore après plusieurs années d'existence. Dans un cadre pédagogique, on étudiera un objet statique pour observer sa coupe tandis qu'une métamorphose permettra de comprendre la nage d'un poisson ou la pousse d'une feuille. Enfin, des calculs peuvent permettre de connaître le moment propice à la coupe d'un arbre.

Figure 1 : Tournesol modélisé avec le logiciel XFrog [Lintermann & Deussen 1999]

Notre intérêt se porte sur la simulation de métamorphoses car, contrairement aux autres applications de l'animation, ce domaine présente la particularité de ne disposer d'aucun système de description interactif performant. Il est en effet extrêmement difficile de per-

mettre à un simple utilisateur de définir interactivement les modifications structurelles successives qui affectent un objet naturel au cours de son évolution.

Ce chapitre a pour objet l'étude des principales techniques existant pour la simulation d'animations. Elle permettra notamment au lecteur de se familiariser avec ce domaine et d'en définir les termes de vocabulaire courants. D'autre part, nous montrons ici pourquoi seules les méthodes basées sur la topologie permettent de décrire et de représenter de manière satisfaisante les modifications structurelles qui définissent la métamorphose d'un objet naturel structuré. Enfin, nous détaillons une étude de cas mettant en évidence les structures et les opérations intervenant dans de telles simulations.

2 Principes généraux et méthodes

Dans cette partie, nous nous attachons à présenter les principales méthodes d'animation existantes en laissant volontairement de côté les méthodes « à base topologique » que nous étudions de manière plus approfondie dans la partie suivante. L'étude présentée ici, inspirée de celle de [Terraz 1994], a pour but de familiariser le lecteur avec le vocabulaire et les différentes techniques d'animation. Pour ça, nous avons choisi d'introduire les méthodes selon deux critères : le système de contrôle, qui détaille comment l'utilisateur maîtrise l'animation, et le modèle d'animation, qui introduit des informations propres à un domaine particulier afin d'automatiser le contrôle de l'animation. D'autre part, nous montrons ici pourquoi ces techniques d'animation ne sont pas adaptées à la modélisation des transformations structurelles qui sont le propre des méthodes basées sur la topologie.

2.1 Systèmes de contrôle

Les systèmes de contrôle sont des techniques générales, indépendantes de tout domaine d'application et reposant sur un contrôle de bas niveau, dans le sens où ils impliquent un effort important de conception de la part de l'utilisateur. [Zeltzer 1985] en identifie trois : les systèmes **dirigés**, les systèmes **niveau « animateur »** et les systèmes **niveau tâche**.

2.1.1 Définitions

Plusieurs termes venant de la littérature de l'animation par ordinateur permettent de caractériser précisément une méthode d'animation. Nous aurons recours à ce vocabulaire dans la suite de cette étude et nous proposons donc quelques définitions essentielles pour la bonne compréhension du chapitre. D'après [Terraz 1994] :

> ***Mouvement ou comportement adaptatif*** *: un mouvement est dit adaptatif s'il prend en compte l'environnement extérieur dans lequel évolue l'objet.*
>
> ***Abstraction structurelle*** *: il s'agit de la possibilité, pour l'utilisateur, de définir pour un objet donné les éléments rigides, les éléments déformables, les liens entre les différents éléments, etc.*

***Abstraction procédurale** : il s'agit de la possibilité de définir des mouvements indépendants de la structure des objets manipulés.*

***Abstraction fonctionnelle** : il s'agit de la possibilité de regrouper un élément et un ensemble de procédures pour effectuer un mouvement particulier.*

***Abstraction caractérielle et modélisation environnementale** : définition de caractères dont les objets de la structure pourront hériter et modélisation de l'environnement (au niveau des interactions, des dépendances, etc.) dans lequel les objets évoluent.*

2.1.2 Systèmes dirigés

Dans un **système « dirigé »,** l'utilisateur fournit un certain nombre d'informations et le système déduit lui-même l'animation correspondante grâce à des règles pré-établies, comme par exemple des lois d'interpolation. Cette technique est à mettre en parallèle avec celle de la programmation par contrainte où le système infère des informations à partir de comportements prédéfinis et de données apportées par l'utilisateur. Plusieurs méthodes reposent sur un système dirigé : les interpolations par images clefs, les animations par cinématique inverse ou encore celles basées sur des déformations de l'espace.

2.1.2.1 Interpolation par images clefs

Cette technique consiste à laisser à l'utilisateur le soin de définir une séquence d'images clés ou de positions clés des différents objets. Il doit ensuite choisir des fonctions d'interpolation pour décrire, d'une part, la façon dont varient les paramètres et, d'autre part, la vitesse avec laquelle ceux-ci évoluent. Le système déduit seul les informations manquantes (Figure 2). Cette méthode présente l'intérêt de permettre de définir une animation à partir d'informations limitées, mais elle oblige l'utilisateur à choisir de manière opportune les moments clefs de son animation, et surtout à déterminer précisément la loi générant les images intermédiaires. Ce procédé n'est pas envisageable pour la simulation de métamorphoses car il ne peut pas représenter les modifications structurelles d'un objet. En effet, celles-ci sont de nature plus complexe : par exemple l'ajout d'un élément peut être conditionné à un ensemble de facteurs, de conjonctions de relations, etc. Ce qui n'est pas toujours exprimable par une fonction mathématique.

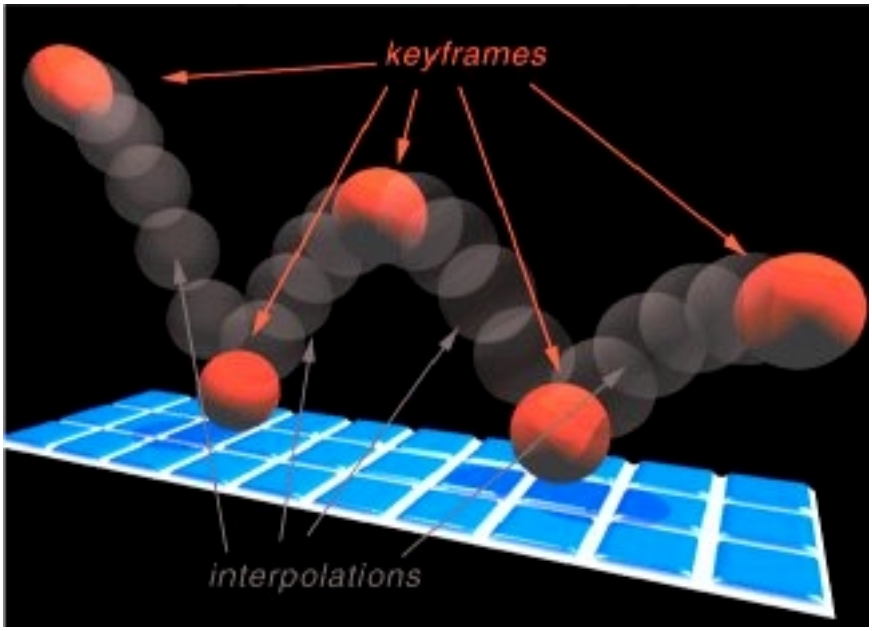


Figure 2: Animation par interpolation de positions clefs

2.1.2.2 Animation de corps articulés par cinématique inverse

Dans le domaine de la modélisation de corps articulés, on veut pouvoir définir le mouvement d'une partie du corps en définissant la trajectoire que l'on veut lui voire suivre. La cinématique inverse répond à ce besoin en appliquant, à partir d'une trajectoire définie par l'utilisateur, une loi capable de déduire la valeur des angles des différentes parties du corps rigides.

Encore une fois, pour les mêmes raisons que précédemment, la méthode de cinématique inverse est parfaitement adaptée au besoin particulier de l'animation de corps rigides mais elle est confinée à la représentation de modifications géométriques et n'est pas adaptable à la modélisation de modifications structurelles.

2.1.2.3 Déformation de l'espace

Une autre technique de définition d'animation par système dirigé consiste à plonger l'objet à animer dans un maillage auquel on applique des déformations (Figure 3). L'avantage de cette méthode est la facilité de redéfinition des différents points de contrôles. Malheureusement, en plus des problèmes inhérents aux systèmes dirigés, on se trouve confronté à une difficulté supplémentaire qui est la définition du maillage. Celle-ci peut en effet devenir très vite difficile dès que l'objet est complexe. Enfin, un tel procédé ne peut être envisagé pour modéliser des modifications topologiques. Il ne s'agit ici que de déformations s'appliquant à la forme de l'objet et en aucun cas à sa structure interne.

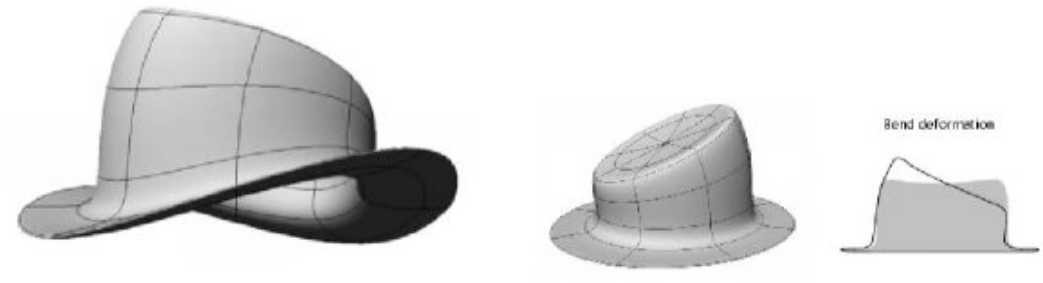


Figure 3 : Déformation par application d'un maillage à un objet

2.1.2.4 Conclusion

Cette solution de contrôle d'animation correspond exactement, en terme de programmation, à la programmation par contraintes. Dans ce type de programmation, le système est capable de générer un programme à partir de règles données et de dépendances définies par l'utilisateur.

Les systèmes dirigés définissent des méthodes adaptées à certains problèmes mais pas de mécanismes de contrôle de haut niveau. L'utilisateur est confronté à des difficultés qui semblent hors de portée d'un non informaticien, comme la définition d'un maillage complexe ou d'une fonction d'interpolation précise. D'autre part, il doit spécifier à l'avance tous les détails de l'animation, ce qui n'est possible que dans un environnement simple. Du point de vue de la définition de métamorphoses, ces méthodes ne sont pas intéressantes dans la mesure où elles ne permettent pas de prendre en compte des modifications topologiques.

2.1.3 Systèmes niveau « animateur »

Dans les **systèmes niveau « animateur »**, l'animation est définie de manière algorithmique. Ces systèmes peuvent avoir recours à des abstractions et à des comportements adaptatifs. L'utilisateur écrit un script qui est ensuite interprété par le système de manière déterministe. On peut citer parmi les plus connus, le vieux mais représentatif système GRAMPS [O'Donnell & Olson 1981] qui permet par exemple de définir à la fois le script associé aux objets et les objets manipulés. Ces systèmes, en autorisant la définition d'objets et de comportements adaptatifs, sont des outils puissants pour la définition d'animations complexes. Mais le fait qu'ils procurent la puissance d'un langage de programmation classique implique aussi qu'ils laissent à l'utilisateur le soin de régler tous les problèmes inhérents au développement logiciel. Leur généralité permet de modéliser une importante catégorie de mouvements ou de déformations mais souvent au prix d'un effort de conception très important de la part de l'utilisateur.

Certains langages utilisent des actions et des objets de granularité moins fine qui simplifient ainsi le rôle de l'utilisateur mais offrent un spectre d'animations réalisables moins large. Etant donné que l'utilisateur ne veut pas toujours spécifier l'animation au niveau le

plus bas (c'est-à-dire préciser toutes les opérations qui seront appliquées à tous les objets de base), différents moyens ont été développés pour lui fournir un contrôle de haut niveau lui permettant de décrire l'animation en termes plus généraux. La description devient ainsi succincte, rapide et souvent compréhensible mais aux dépens d'une certaine précision dans le contrôle de l'animation. Ces moyens sont par exemple :

- La paramétrisation. On définit des paramètres dont les valeurs contrôlent la configuration ou le mouvement de l'objet modélisé. Le problème vient de la difficulté de définir, a priori, des paramètres cohérents pour le plus grand nombre d'utilisations possibles. La gestion de ces paramètres est un des points importants que nous aborderons lorsque nous présenterons la programmation sur exemple.
- La hiérarchisation des commandes, qui consiste à définir une structuration de plus en plus détaillée de l'objet à animer puis à associer des commandes aux sous-objets en fonction de leurs niveaux respectifs (par exemple, un bras est décomposé en épaule, coude, avant-bras, main ; une main en paume et doigt. La commande « plier bras » se décompose alors en « plier coude », « plier main » et ainsi de suite jusqu'aux éléments de base).
- Les librairies de commandes, qui regroupent des actions de bas niveau sous des termes plus généraux et rendent leur utilisation plus facile. Ceci étant, elles restent difficiles à assembler (certains ordres pouvant être involontairement contradictoires).

L'approche niveau « animateur » correspond, en terme de programmation, à une technique impérative dans laquelle le programmeur exerce un contrôle total sur son programme, contrairement au cas de la programmation par contraintes.

Le fait qu'elle soit si proche de la programmation rend cette approche très puissante et permet d'envisager la représentation de tout type de modifications, qu'elles soient géométriques ou topologiques. Elle se trouve de ce fait totalement adaptée à la simulation de métamorphoses.

2.1.4 Systèmes niveau tâche

Dans les systèmes de ce type, l'animation est générée à partir d'informations limitées : l'utilisateur fournit la position initiale de l'objet, les forces qui s'appliquent sur cet objet, etc. L'animation est ensuite déduite de ces informations par l'application de lois physiques par exemple. Cette technique permet de prendre en compte un domaine particulier et est donc destinée à des applications précises.

Ce type de système d'animation n'est pas adaptable à la description d'une métamorphose. Il n'est pas possible de décrire les modifications structurelles complexes, prenant en compte un nombre important de paramètres et de conditions, par des lois.

2.1.5 Conclusion

Les systèmes décrits ci-dessus, y compris ceux de niveau « animateur » utilisant un contrôle de haut niveau, requièrent un effort important de la part de l'utilisateur pour la définition d'une animation, que ce soit pour déterminer un maillage, une loi mathématique représentant un comportement ou un algorithme basé sur une syntaxe plus ou moins rébarbative. Parmi ces différentes techniques, on remarque cependant que seules celles de niveau « animateur » peuvent prétendre décrire les modifications structurelles propres à la modélisation d'une simulation de métamorphose.

Les difficultés posées par ces méthodes ont amené les auteurs à déterminer des modèles faisant appel à des connaissances issues d'autres disciplines et permettant un contrôle plus automatique de l'animation. Le paragraphe suivant décrit brièvement en quoi consistent ces différents modèles.

2.2 Modèles d'animation

Le principal problème soulevé par l'utilisation des systèmes de contrôle vient de l'effort de conception trop important demandé à l'utilisateur. Les modèles d'animation sont des techniques qui tendent à résoudre ou au moins à diminuer cette difficulté en faisant appel à des connaissances d'autres domaines. Qu'ils soient descriptifs, générateurs ou comportementaux, ils sont tous destinés à apporter une aide à l'utilisateur dans le cadre d'une application à l'univers bien défini. Celui-ci n'a en effet aucun moyen de modifier ou de raffiner le modèle pour le faire correspondre exactement à un besoin plus large ou au contraire plus précis. Leur seule utilité est bien d'essayer de soulager la charge de l'utilisateur, au détriment de son pouvoir d'expression. Ainsi, certains modèles sont basés sur des lois physiques, d'autres sur un schéma de comportement humain ou animal, d'autres encore sont basés sur des règles de botanique, etc.

La décomposition en trois groupes (descriptif, générateur et comportemental) que nous présentons par la suite est inspirée de [Hegron & Arnaldi 1992].

2.2.1 Modèles descriptifs ou phénoménologiques

Les modèles dits descriptifs ou phénoménologiques s'appliquent à simuler un effet sans aucune connaissance de sa cause. Ils peuvent être basés sur un contrôle stochastique, comme c'est le cas par exemple dans la réalisation d'images fractales ([Mandelbrot 1975]) très utilisée dans la représentation de paysages (montagnes, forêts) ou d'animations. Ils peuvent également utiliser un contrôle morphologique permettant une simulation très simple, fondée sur l'aspect régulier de certains objets (comme les plantes).

Ces modèles obtiennent des résultats très spectaculaires au niveau du réalisme mais ne peuvent pas être utilisés pour représenter des simulations de métamorphoses. Par exemple, la modification d'une image fractale entraîne sa redéfinition complète et il n'est donc pas concevable d'obtenir un contrôle précis sur une modélisation de ce type dans le cadre d'une animation.

2.2.2 *Modèles générateurs ou physiques*

Dans les modèles générateurs, la modélisation des objets se fait selon leurs propriétés physiques. On distingue ainsi généralement les objets rigides, les objets déformables et les objets dits « mixtes », présentant des caractéristiques rigides et déformables. La méthode générale d'animation consiste ensuite à décrire les forces qui s'appliquent sur un objet donné, mais elle diffère d'une modélisation à l'autre. La suite de cette partie présente les particularités liées à chaque type d'objet.

2.2.2.1 Objets rigides

Les objets sont définis par un ensemble de solides rigides, caractérisés par des propriétés physiques (masse, inertie, etc.). Des liens définissant le degré de liberté de ces solides les uns par rapport aux autres les unissent (Figure 4).

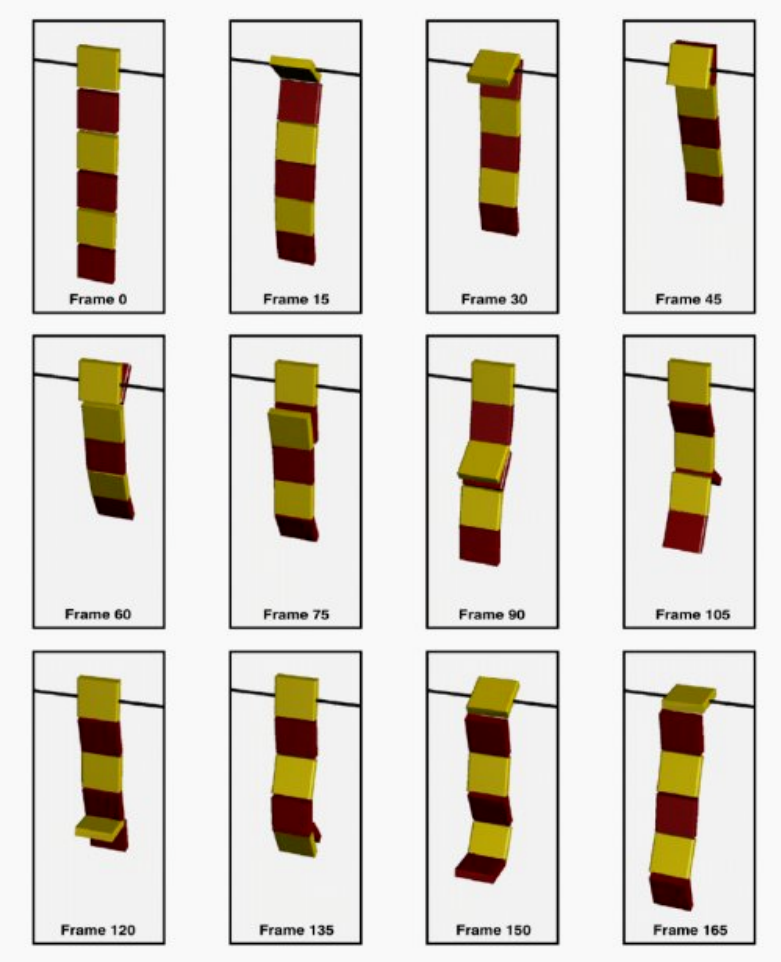


Figure 4 : Animation dirigée par des lois physiques d'un objet rigide composé de plusieurs éléments

Plusieurs méthodes coexistent pour définir l'animation de tels objets :

- L'utilisateur fournit au système l'état initial de l'objet ainsi que toutes les forces à appliquer à chaque partie de l'objet. Cela implique que l'utilisateur connaît tous les paramètres physiques de l'animation à réaliser.

- Une autre méthode utilise la dynamique et la dynamique inverse. Dans ce cas, l'utilisateur définit un certain nombre de contraintes géométriques et d'informations sur l'objet (position initiale, vitesse initiale, accélération, etc.). Le système calcule alors lui-même les différentes forces à appliquer pour que les contraintes géométriques soient respectées tout au long de l'animation.
- Une méthode plus récente repose sur l'utilisation de moments « clefs ». L'utilisateur doit être en mesure d'informer le système sur des paramètres (position, vitesse, etc.) à certaines étapes critiques de l'animation. Le système génère ensuite seul les étapes intermédiaires de la simulation en s'assurant que l'animation reste cohérente d'un point de vue physique.

2.2.2.2 Objets déformables

Par opposition aux objets rigides, on étudie les objets déformables au niveau de la composition de leur matériau et on s'intéresse donc à leur déformation en fonction de leur élasticité, de leur dureté, etc.

Pour simuler ces déformations, on peut ou bien faire appel à des équations mécaniques complexes, ou bien utiliser une modélisation originale des objets. Une méthode consiste à simuler un objet déformable par un ensemble d'éléments mécaniques discrets connectés entre eux par des liaisons de type ressort. Cette technique a l'avantage d'être très simple à mettre en place, d'être plus facile à contrôler et de donner des résultats spectaculairement naturels.

2.2.2.3 Objets mixtes

L'approche mixte, comme son nom l'indique, prend en compte les deux aspects des méthodes précédemment citées en « habillant » un modèle rigide articulé par un modèle déformable. Ainsi, un personnage peut être modélisé par un squelette et une couche déformable (Figure 5).

Figure 5 : un exemple de modèle mixte : le personnage principal du film d'animation « Shreck » (Studio Dreamworks™)

2.2.2.4 Conclusion

Comme nous l'avons expliqué, ces méthodes bénéficient de l'apport de connaissances du domaine pour restreindre le champ d'application et faciliter la définition de mouvements. Ceci étant, aucune de celles présentées dans ce paragraphe ne permet de prendre en

considération l'évolution topologique d'un objet. Que ce soit la dynamique inverse, l'utilisation de moments clefs ou encore l'étude des déformations d'un objet en fonction de sa composition, aucune méthode n'est en mesure d'être utilisable dans le contexte de la description des modifications structurelles apportées à un objet.

2.2.3 Modèles comportementaux

D'après ([Gwenola & Donokian 2000]), le but des modèles comportementaux est de simuler le comportement de toutes sortes d'individus vivants (plantes, animaux et êtres humains). Les auteurs classifient ainsi les modèles comportementaux :

- Les modèles de transformation interne, provoquant des changements externes de l'objet (croissance de plantes, muscles, etc.). Nous reviendrons dans la partie 3 de ce chapitre sur la méthode de l'AMAP dont la modélisation de plante utilise des connaissances en botanique ;
- Les modèles d'animation externe, définissant le comportement extérieur d'un être, ses actions et ses réactions, de manière individuelle (animal, humain) ou collective (nuée d'oiseaux, banc de poissons, troupeau de mammifères). Par exemple, le vol d'un groupe d'oiseaux peut-être modélisé de cette façon, en supposant qu'il est le résultat de l'interaction entre le comportements des différents oiseaux respectant un certain nombre de règles (éviter les collisions, égaler la vitesse des voisins, etc.) ;

Les modèles d'animation externe sont tous fondés sur un système relationnel entre des acteurs et leur environnement ce qui implique deux types de relations : la perception et l'action. Plusieurs approches ont été étudiée pour la définition du modèle décisionnel : stimulus / réponse, règles de comportement, environnement prédéfini et automates. Le constat que l'on peut faire de ces différents travaux est que ce sont des modèles ad hoc conçus pour être appliqués à chaque fois sur des cas particuliers, dans lesquels les objets et leur environnement sont relativement simples et les champs de perception et d'action limités. Il existe en outre une grande disparité dans les comportements permis par chacune des approches :

- Soit le niveau d'abstraction est très faible et seuls des comportement de type réflexes pourront être spécifiés (approche stimulus / réponse),
- Soit le niveau d'abstraction est plus élevé mais l'environnement est réduit, voire complètement défini, et donc la perception et l'action de l'entité sont d'un niveau très faible.

Une autre différence réside dans les buts recherchés : cela varie de la définition d'entités complètement autonomes dont le comportement est relativement ciblé et difficilement « spécifiable » (problème du choix des paramètres), à des entités, évoluant dans un environnement bien défini, dont le comportement est quasiment complètement spécifié par un scénario.

Cette description montre bien que les modèles comportementaux possèdent des caractéristiques compatibles avec la modélisation de métamorphoses. Par exemple, les modèles de transformation interne peuvent être utilisés pour simuler l'activité des bourgeons d'un arbre (qui peuvent mourir, se ramifier, etc.) en fonction d'événements dictés par des lois spécifiques à l'espèce et à la variété. Dans le même ordre d'idée, les modèles d'animation externe peuvent servir à prendre en compte le taux d'hydrométrie, le degré d'ensoleillement ou encore la fertilité du sol.

2.3 Logiciels d'animation commerciaux

Il est important de noter que tous les logiciels commerciaux récents d'animation intègrent la plupart des techniques décrites ici. On peut citer entre autres Maya™ ([Aliaswavefont 2002]), Softimage™ ([Softimage 2002]) ou 3D Studio™ ([Discreet 2002]). Ce dernier (Figure 6), par exemple, dispose des outils suivants :

- SET-KEY ANIMATION SYSTEM : système intégré d'interpolation par positions clefs ;
- SKIN POSE : permet de définir pour un personnage une posture de maillage « Skin Pose » ;
- SPLINE IK : solveur de chaîne de cinématique inverse ;
- DYNAMICS Discreet reactor : permet le calcul des solutions dynamiques interactives et en temps réel avec les notions physiques prédéfinies pour simuler des comportements propres à des corps rigides (« Hard Body »), déformables (« Soft Body »), ou à des schémas encore plus particuliers correspondants à des vêtements ou à des surfaces fluides.

Figure 6 : Extrait d'une animation de personnages réalisée avec 3DStudio

2.4 Conclusion

Au cours de cette partie, nous avons balayé de manière rapide l'ensemble des techniques permettant de réaliser des animations d'objets par ordinateur. Ce qui ressort de cette étude est que la plupart des efforts entrepris dans le domaine concernent les mouvements et la déformation. Les systèmes dirigés ou niveau tâches ainsi que les modèles générateurs ou descriptifs ne s'intéressent ainsi qu'aux modifications des attributs géométriques ou à la déformation de ce qui représente l'enveloppe des objets modélisés. Seuls finalement les systèmes niveau « animateur » et les méthodes comportementales laissent la porte ouverte à la modélisation de métamorphoses en permettant des modifications topologiques de l'objet étudié.

A la lecture de ces quelques pages, il ressort également un parallèle intéressant entre la programmation et les différentes méthodes proposées. En effet, le premier groupe de méthodes, n'autorisant pas les modifications structurelles, peut-être comparé à l'approche déclarative de la programmation dans laquelle l'utilisateur fournit des données de départ à un système qui, à partir de règles pré-établies, déduit des informations en conséquence. Les systèmes « animateur » et les méthodes comportementales peuvent elles être comparées à une approche impérative, dans laquelle l'utilisateur décrit séquentiellement les actions à effectuer.

Une autre constatation, analysée par [Terraz 1994] est que le modèle, qu'il soit générateur, descriptif ou comportemental, ne suffit pas à donner à une méthode une grande puissance d'expression. Celle-ci dépend également du niveau de description et du degré de précision du contrôle. On peut par exemple utiliser un système niveau « animateur » avec un langage de très haut niveau et donc limiter la puissance d'expression de la méthode. Tout comme dans tout système informatique, il est difficile de concilier puissance d'expression et contrôle de haut niveau : plus la granularité des actions augmente, plus la facilité d'utilisation augmente et plus le degré de contrôle diminue.

La partie suivante présente les méthodes spécifiques à la modélisation de métamorphoses d'objets naturels structurés.

3 Métamorphoses d'objets naturels

3.1 Introduction

En marge des travaux présentés dans la partie précédente, il existe des méthodes d'animation spécialisées dans le domaine de la simulation d'évolution d'objets naturels, comme la croissance d'une plante ou la nage d'un poisson. Les L-Systems, par exemple, utilisent des grammaires constituées de règles de réécriture permettant d'associer à un certain schéma une évolution type ; les méthodes « à base topologique » reposent elles sur une décomposition d'un objet en un ensemble de sous-objets possédant chacun un comportement évolutif propre.

Dans cette partie, nous présentons plusieurs méthodes permettant de représenter les différentes étapes du processus d'évolution d'un objet naturel. Contrairement à celles exposées dans la partie précédente, celles-ci présentent la particularité de proposer une modélisation et un système de contrôle permettant de décrire précisément les modifications structurelles des objets. Cette étude nous permet de mettre en évidence les principales caractéristiques de ces méthodes et de comprendre dans quelles mesures celles basées sur la topologie offrent un champ d'action privilégié pour mettre en place un système interactif d'aide à la création de métamorphoses.

3.2 Systèmes basés sur l'utilisation de grammaires

Originellement, les systèmes basés sur l'utilisation de grammaires, les L-systèmes, ont été introduits pour modéliser le développement d'organismes multicellulaires simples (par exemple une algue) en terme de division, croissance et mort des cellules individuelles. Le champ d'application des L-systèmes a été étendu à de plus grandes plantes et à des structures d'arbre plus complexes, en particulier la floraison, décrite comme étant la formation de modules dans l'espace. Dans le contexte des L-systèmes, le terme module symbolise toute unité de construction discrète qui se répète quand la plante se développe (par exemple un entrenœud (ou arête), un nœud, une fleur ou une branche). Le but de la modélisation au niveau modulaire est de décrire le développement d'une plante dans sa globalité (en particulier au niveau de sa forme) aussi bien que dans l'intégration du développement de ses unités individuelles.

3.2.1 Principes généraux

Comme l'explique [Prusinkiewicz & Hammel 1996], l'essentiel du développement au niveau modulaire peut être réalisé facilement par un système de réécriture parallèle : celui-ci remplace des modules parents (encore appelés mères ou ancêtres) par des configurations de modules enfants (ou filles ou descendants). Tous les modules appartiennent à un alphabet de modules fini, ainsi un grand nombre de configurations de modules peut être spécifié en utilisant un ensemble fini de règles de réécriture (ou de productions). Dans le cas le plus simple, une production est un module simple appelé prédécesseur (ou partie gauche) et une configuration de zéro, un ou plusieurs modules est appelé successeur (ou partie droite). Une règle de production p dont le prédécesseur concorde avec un module mère peut être appliqué en effaçant ce module de la structure réécrite et en insérant le module fille spécifié par le successeur de p. Les Figure 7 et Figure 8 montrent deux exemples de règles de production et de leur application. Dans le premier cas, les modules localisés aux extrémités de l'arbre sont remplacés sans affecter le reste de la structure : un bourgeon donne une fleur, qui produit un jeune fruit, lequel dépérit en vieux fruit. Dans tous les cas, la modification s'opère sur la partie terminale d'un axe, la structure n'en subit donc aucune conséquence. Dans l'autre cas, les règles de productions qui remplacent les entrenœuds divisent l'arbre en une partie basse et une partie haute (avant et après l'entrenœud). La position de la partie haute est recalculée en conséquence mais la taille et

la forme des deux parties ne sont pas modifiées. La différence par rapport à l'exemple précédent est que la modification s'opère sur une partie de l'arbre qui n'est pas terminale : elle entraîne donc un changement dans la position spatiale des éléments placés en aval.

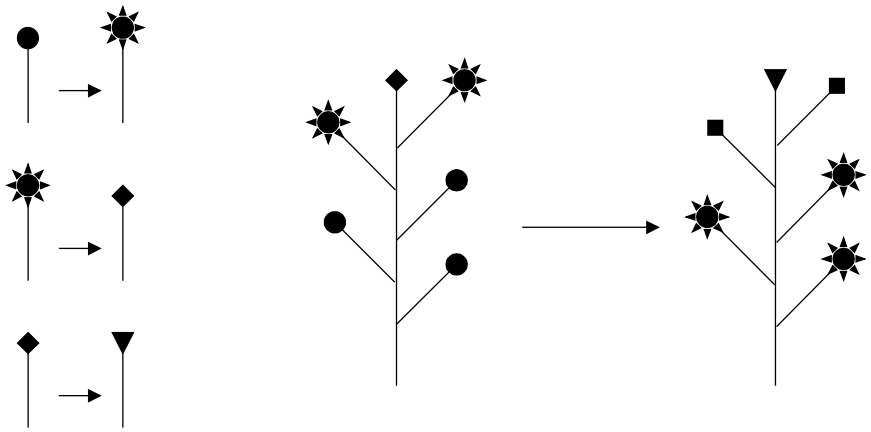


Figure 7 : Exemple de spécification de règles de production et de leur application sur un exemple

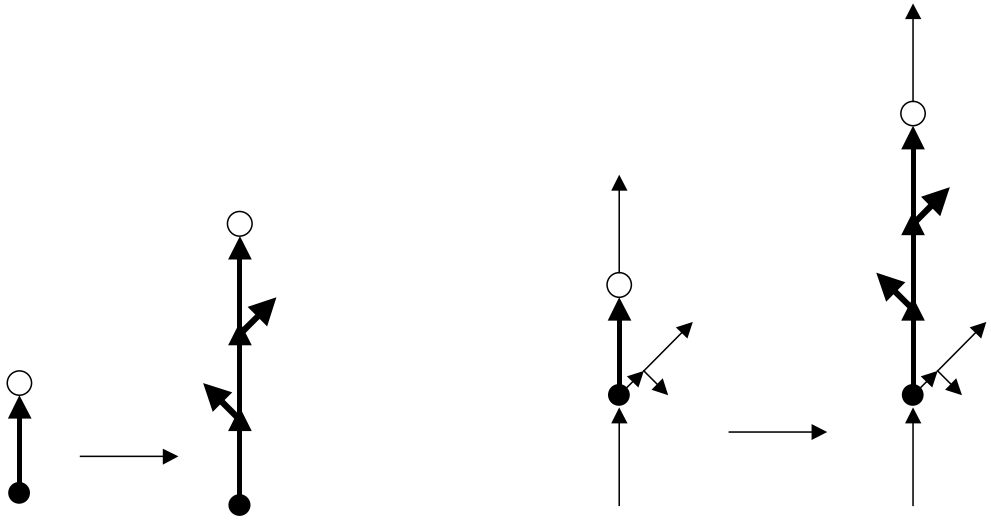


Figure 8 : Exemple de spécification d'une règle de production et de son application sur un exemple

Les règles de production peuvent être appliquées séquentiellement (un module à la fois) ou en parallèle (tous les modules simultanément, à chaque pas de dérivation). Cette deuxième méthode est plus adaptée pour la modélisation de développements d'ordre biologiques, puisque la croissance a lieu dans toutes les parties de l'organisme en même temps. Une séquence de structures obtenue à partir d'une structure initiale (appelée aussi axiome) par plusieurs pas de dérivation successifs est appelée séquence de développement (cela correspond au résultat d'une simulation de développement dans laquelle le temps est discrétisé). Par exemple, la Figure 9 illustre le développement d'une feuille stylisée composée de deux types de modules : les arêtes terminales (lignes fines) et les entrenœuds (lignes épaisses). Une arête terminale produit une structure composée de deux entrenœuds, deux arêtes terminales latérales et une réplique de l'arête terminale initiale.

Un entrenœud, lui, s'allonge par un facteur constant. En dépit de leur simplicité, ces règles développent une structure d'arbre complexe à partir d'une seule arête terminale.

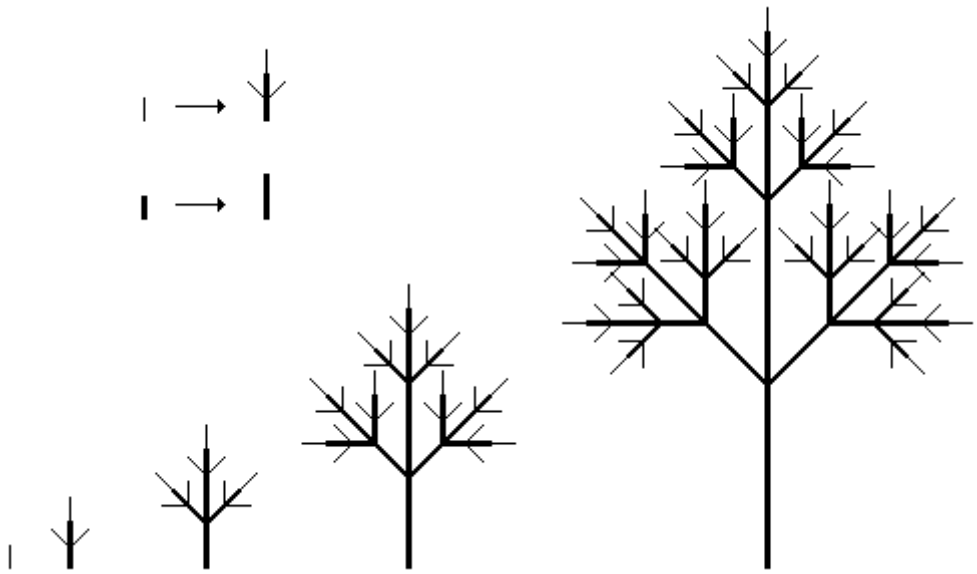


Figure 9 : Développement à partir de deux modules et de deux règles de production

Les méthodes basées sur des grammaires permettent de représenter les modifications topologiques qui affectent un objet au cours de son évolution. Ceci étant, la description graphique des règles n'en facilite pas l'utilisation. Dans la paragraphe suivant, nous montrons comment il est possible, grâce à une notation particulière, d'exprimer des lois de cette nature dans un formalisme plus facile à manipuler.

3.2.2 Notation à base de chaînes de caractères parenthésées

Les systèmes de réécriture pour des métamorphoses d'arbre peuvent être définis directement dans le domaine géométrique (règles à base d'objets géométriques), mais il est plus pratique d'exprimer les règles de production et les structures résultantes de manière symbolique, en utilisant une notation sous forme de chaîne de caractères. Pour cette raison, on utilise une représentation sous forme de chaînes de caractères parenthésées : un arbre dont les arêtes sont étiquetées par des éléments d'un alphabet V est représenté par un mot w sur un alphabet $V_E = V \cup \{[,]\}$:

$$w = x_1[\alpha_1]x_2[\alpha_2]...x_n[\alpha_n]x_{n+1}$$

$$x_i \in V^*, \alpha_j \in V_E^* \text{ avec } i \in [1..n+1] \text{ et } j \in [1..n]$$

On suppose que les sous mots $x_1, x_2, ..., x_{n+1}$ ne contiennent pas de parenthèses et que les sous mots $\alpha_1, \alpha_2, ..., \alpha_n$ sont bien parenthésés. Le mot $x_1x_2...x_{n+1}$ représente l'axe principal (d'ordre 1) de w, dont les entrenœuds sont $x_1, x_2, ..., x_n$ et l'arête terminale x_{n+1}. Les mots

α_1, α_2, ..., α_n représentent les axes d'ordre 2 de w. Chaque branche α_i peut être décomposée de la même façon que w (et ainsi de suite ...). La décomposition d'un mot w bien parenthésé est unique, donc tous les termes introduits ne présentent pas d'ambiguïté.

Par exemple l'arbre présenté dans Figure 10 est représenté par la chaîne de caractères suivante :

$$w = ab\,[cd]\,ef\,[g[h]i]\,j\,[k]\,lm$$

Le mot abefjlm est l 'axe principal de w. x_1 = ab, x_2 = ef, x_3 = j sont les entrenœuds et x_4 = lm est l'arête terminale. Les mots α_1 = cd, α_2 = g[h]i et α_3 = k dénotent les branches latérales. α_1 et α_3 possèdent uniquement des arêtes terminales tandis que α_2 possède un entrenœud g, une arête terminale i et une branche latérale (d'ordre 3) h.

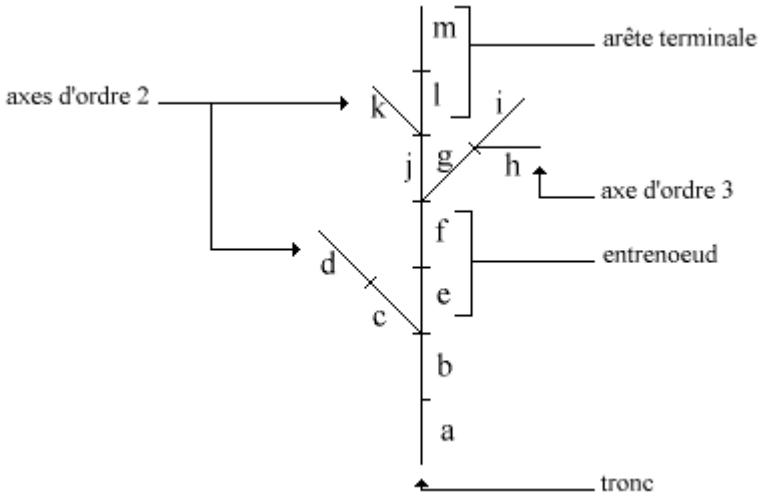


Figure 10 : Arbre correspondant à la chaîne ab[cd]ef[g[h]i]j[k]lm

Les chaînes de caractères paramétrées ont été introduites pour représenter des structures d'arbre. Pour exprimer les propriétés telles que l'orientation des branches et la longueur des arêtes, il faut assigner une interprétation géométrique aux chaînes. Dans les cas simples, les informations géométriques sont extérieures aux chaînes (on peut par exemple considérer que tous les segments produits ont une même longueur, constante).

Mais les règles opérant ainsi, de manière uniforme sur l'ensemble de la chaîne de caractères, ne sont pas suffisantes pour exprimer tous les développements que l'on trouve dans la nature. Pour cette raison, on a recourt à des techniques permettant l'incorporation d'informations géométriques à l'intérieur des chaînes de caractères. Certains symboles, réservés peuvent apparaître avec ou sans paramètres. Ainsi, « the turtle interpretation » ([Szilard & Quinton 1979]) utilise cette approche : une tortue (en référence au LOGO), localisée initialement à l'origine du repère cartésien du système de coordonnées est orientée puis déplacée par un certain nombre de symboles. Par exemple, F (s) déplace la

tortue vers l'avant d'une longueur de s en lui faisant dessiner un segment de sa position initiale à sa position d'arrivée (Figure 11).

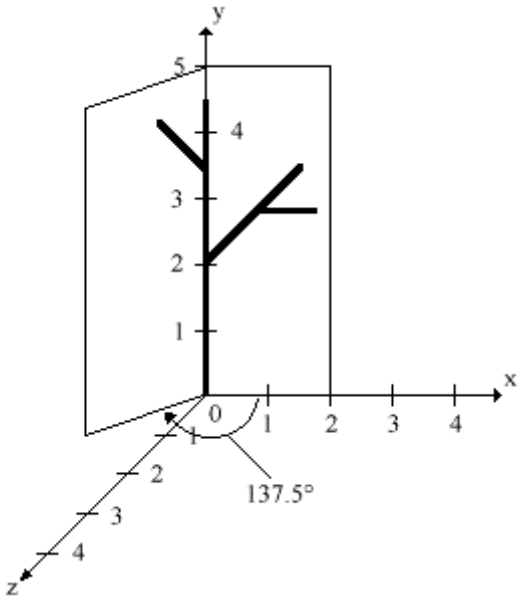


Figure 11 : Exemple de l'interprétation d'une chaîne de caractères avec « the turtle interpretation ». L'arbre correspond à la chaîne :F(2)[-F[-F]F]/(137.5)F(1.5)[-F]F[1]
La longueur des lignes représentée par F sans paramètres est 1 et l'amplitude des angles représentés par + et - est 45°.

3.2.3 Conclusion

Les méthodes basées sur des grammaires présentent l'avantage, a priori attrayant, de mettre en jeu une technique très utilisée en informatique. En outre, nous avons vu au cours de cette partie comment il était possible de transposer les règles de réécriture géométrique dans un formalisme dont la manipulation est plus aisée. L'intérêt de cette méthode, par rapport à un langage de programmation, est la simplicité de description et le recours à un formalisme facile d'accès.

Malheureusement, si le système de règles de réécriture convient parfaitement pour des cas simples, il faut enrichir de manière démesurée le format pour, par exemple, comme l'illustre le paragraphe précédent, être en mesure d'obtenir des développements paramétrés et faire ainsi en sorte que l'évolution d'un objet soit différente d'une exécution à l'autre. La mise en place de concepts tels que les paramètres complexifie la description des règles au point d'en faire un véritable langage de programmation. La méthode perd alors une grande partie de son intérêt initial.

---

[1] F(s) déplace vers l'avant d'une longueur s en traçant un segment.
/ (σ) tourne d'un angle σ autour de l'axe des y.

Les techniques basées sur la topologie, que nous allons étudier maintenant, sont contrôlées de manière « naturelle » par des systèmes niveau « animateur », c'est-à-dire de manière algorithmique.

3.3 Méthodes basées sur la topologie

Pour représenter les métamorphoses d'objets naturels, les méthodes les plus utilisées sont celles basées sur la topologie, car elles reposent sur une modélisation proche de la structure réelle des objets. Celles-ci simulent un phénomène naturel en générant une séquence d'objets (O1, O2, … On), Ot étant l'objet à l'instant t, et t étant un temps discret (une telle séquence est appelée une **métamorphose**). Les principes de base de ces méthodes sont les suivants :

- les objets sont structurés en sous-objets ;
- un comportement est associé à chaque objet ou sous-objet et O_i est déduit de O_{i-1} par application à tous les sous-objets de O_{i-1} de leur comportement associé. Un comportement associé à un sous-objet est en fait la description de l'évolution de ce sous-objet. Pour ces méthodes, les comportements sont des ensembles d'opérations qui seront appliquées à chaque étape au sous-objet associé. Comme un objet est structuré, le contrôle de la construction d'une métamorphose peut être local ou global. De plus ces comportements peuvent être paramétrés et donc dépendants du contexte.

Parmi les aspects importants de ces méthodes, on peut souligner le fait que les objets manipulés sont des subdivisions d'espaces géométriques (par exemple : sommets, arêtes, faces, etc.). Cette partition de l'objet correspond à la structuration de base de l'objet, sur laquelle des sous-objets de plus haut niveau peuvent être définis. De telles organisations sont importantes pour la modélisation d'objets naturels, car la plupart d'entre eux sont déjà naturellement structurés. D'autre part, des opérations de base sont explicitement définies et utilisées pour définir les comportements de ces sous-objets. Ces comportements sont paramétrés et des opérations topologiques peuvent être appliquées pour modifier la composition des objets (la croissance peut être simulée par ajout de nouvelles parties à l'objet). Il faut noter enfin que la réalisation effective de la transformation globale de l'objet correspond simplement à appliquer les comportements des différents sous-objets dans un ordre donné (celui du parcours de l'objet).

Dans la suite de cette partie, nous présentons quelques méthodes basées sur la topologie dont les descriptions sont inspirées de [Lienhardt 1987] et [Terraz 1994]. Il s'agit de comprendre, à partir de ces différentes approches, les principes de fonctionnements des méthodes basées sur la topologie.

3.3.1 Systèmes de particules

Les méthodes à base de systèmes de particules ont été développées par [Reeves 1983] et utilisées par [Reynolds 1987] pour simuler des mouvements de groupes tels que les vols

d'oiseaux et les migrations d'animaux. Il s'agit de méthodes très générales pouvant être appliquées à n'importe quel domaine.

Grossièrement, un système de particules est un ensemble de particules. Les particules sont assimilables à des objets topologiques de dimension 0 auxquels sont associés des points dans le modèle de plongement géométrique (on peut également associer des formes géométriques). Plus précisément, les particules sont organisées hiérarchiquement : un système de particules est soit un ensemble de particules, soit un ensemble de systèmes de particules.

Pour manipuler un système de particules, on dispose de trois opérations de base, deux topologiques (création et suppression des particules) et une géométrique (déplacement des particules).

Un comportement standard est défini pour toutes les particules et un comportement particulier est déterminé en fonctions d'attributs, constants ou variables (dépendants du temps par exemple) associés à la particule ou au système. Chaque particule est caractérisée par un certain nombre d'attributs : durée de vie, position et vitesse initiale à la création, etc. Il en est de même pour le système : nombre de particules créées à chaque étape, forme initiale (qui définit la position des particules lors de la création du système).

Le comportement des particules est défini en utilisant un principe de filiation : un système de particules définit les attributs des particules qu'il génère et donc leur comportement. Le comportement standard d'un système de particules est, lui, défini par une séquence d'opérations de base : création de nouvelles particules, définition de leurs attributs, suppression et déplacement. Chacune des opérations est paramétrisée par les attributs du système.

Les systèmes de particules sont des systèmes généraux, non dédiés à une classe d'applications particulière. Ils sont donc adaptables mais nécessitent en contrepartie un effort d'analyse préalable important de la part du concepteur. Le contrôle de la description d'une métamorphose par ce type de méthode se fait en règle général par un système niveau « animateur », c'est-à-dire de manière algorithmique. On peut également introduire des connaissances du domaine étudié en utilisant un modèle comportemental. Par exemple, le vol d'un groupe d'oiseau peut être décrit pas un ensemble de règles régissant le comportement d'un individu du groupe : « Eviter les collisions », « Egaler la vitesse de son voisin ». Le comportement général (Position, orientation et but du vol) est défini comme paramètre global du système.

3.3.2 AMAP

L'AMAP (Atelier de Modélisation d'Architecture de Plantes) s'est constitué à la suite des travaux de De Reffye ([Reffye, et al. 1988]), réunissant botanistes, agronomes et informaticiens, avec pour objectif de modéliser l'architecture des plantes et de simuler leur croissance sur ordinateur.

Les objets modélisés sont des arbres, découpés en cellules de dimension 0 et 1. Un arbre est structuré de la même manière qu'un arbre naturel (Figure 12) :

- un sommet de l'arbre correspond à un nœud de l'arbre naturel, c'est-à-dire à la position d'insertion des feuilles et des bourgeons ;
- une arête correspond à un entrenœud, c'est-à-dire à la partie de l'arbre comprise entre deux nœuds consécutifs ;
- les sommets et les arêtes sont structurés en unités de croissance, ce qui correspond au fait que la croissance des arbres se fait en deux étapes distinctes. Premièrement, une série d'entrenœuds est produite dans un bourgeon (généralement dans un temps très court), définissant une première unité de croissance. Ensuite, la taille des entrenœuds augmente (généralement pendant un temps plus long) ;
- les unités de croissance sont structurées en axes, correspondant à la notion botanique d'axes : le tronc est l'axe principal (axe d'ordre 1), sur lequel les branches principales sont insérées (axes d'ordre 2), etc.

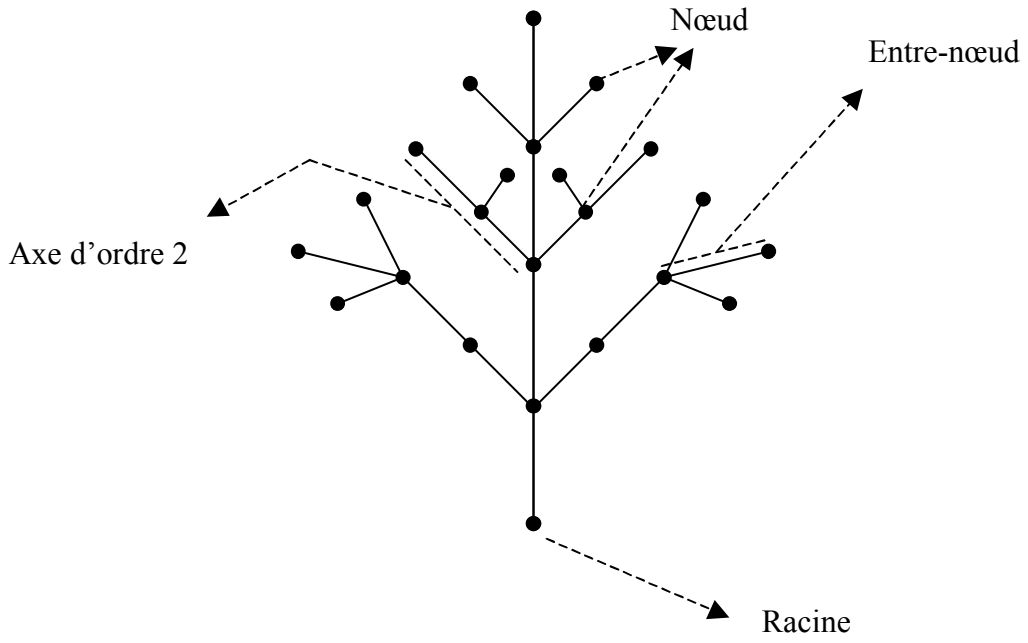


Figure 12 : Les différents composants d'un arbre naturel

Le principe de la méthode consiste à simuler l'activité des bourgeons. Ceux-ci peuvent :

- mourir lors de l'apparition d'une fleur ou lors d'un traumatisme ;
- croître, immédiatement ou suivant un certain délai ;
- se ramifier : ce comportement correspond à l'existence de bourgeons auxiliaires, qui peuvent produire des axes d'ordre supérieur, des réitérations, ou des organes comme les fleurs ou les palmes. Une réitération est une reproduction totale ou partielle de la plante (schématiquement, ceci correspond à un comportement récursif). Les réitérations apparaissent souvent sur des arbres vieux, ou ayant subi un traumatisme. Les différents types de ramification sont les ramifications continues (chaque nœud d'un axe est l'origine d'un axe d'ordre supérieur), rythmiques

(certains nœuds précis sont origine d'un axe d'ordre supérieur), et diffuses (les nœuds origines sont répartis aléatoirement).

Ces événements ont lieu en fonction de lois spécifiques à l'espèce et à la variété d'arbre : il s'agit d'un modèle comportemental de transformation interne qui prend en compte des connaissances propres au domaine. Les paramètres utilisés dans le processus de simulation sont : l'âge de la plante, la vitesse de croissance d'un axe, les probabilités décrivant l'activité d'un bourgeon (qui sont fonction de l'âge et de l'ordre du bourgeon, par exemple), le nombre d'entrenœuds par unité de croissance, etc. Les paramètres géométriques, comme la longueur et le diamètre d'un entrenœud, la phyllotaxie (position des bourgeons auxiliaires sur un axe), ou les angles de branchement sont aussi calculés en fonction de lois spécifiques (un arbre est plongé hiérarchiquement). D'autres phénomènes peuvent être simulés, par exemple la chute de branches ou leur fléchissement en fonction de lois de résistance des matériaux.

Différents types de croissance peuvent être simulés en utilisant ces paramètres (Figure 13 et Figure 14).

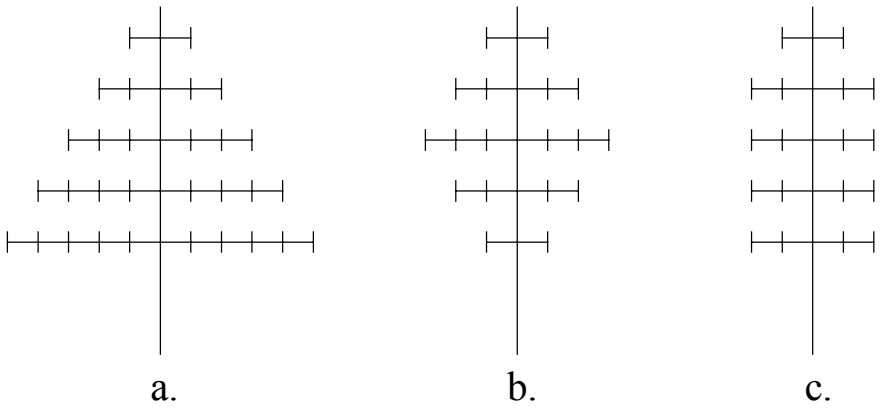


Figure 13 : Différents développements d'axes secondaires
a) l'axe secondaire croît de manière illimitée
b) la taille d'un axe (en terme de nombre d'arêtes) dépend de la distance entre l'origine de cet axe et la racine de l'arbre
c) la taille de l'axe secondaire est limité à un étage fixe..

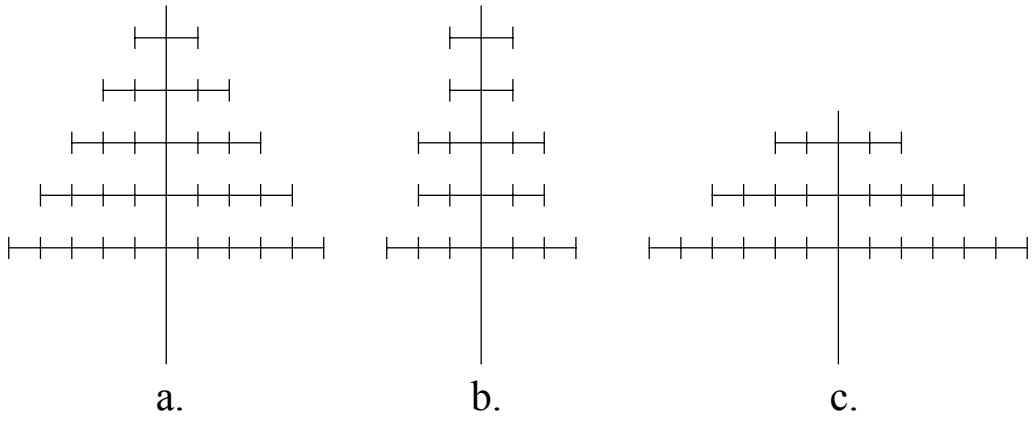


Figure 14 : Différentes vitesses de développements des axes secondaires
a) Identique à celle du tronc
b) Deux fois inférieure à celle du tronc
c) Deux fois supérieure à celle du tronc

Les probabilités et les paramètres sont mesurés sur les plantes réelles, et les simulations valident le modèle. Les expérimentations ont révélé certains phénomènes botaniques, ayant comme conséquences la définition de nouveaux concepts et la simplification de la méthode. Par exemple, un nouveau concept est l'âge physiologique d'un bourgeon. Il a entraîné la définition de la notion d'axe de référence, qui décrit tous les états d'un bourgeon en fonction de son âge physiologique. La notion d'axe de référence explique de nombreux phénomènes comme la ramification, qui correspond à une transition vers un âge plus élevé, et la réitération, qui consiste à garder le même âge.

L'AMAP est une méthode attachée à un domaine très précis, la croissance des plantes. De ce fait, l'utilisateur n'a pas à modifier le modèle pour pouvoir l'utiliser. D'autre part les résultats obtenus par ces méthodes sont spectaculaires et les représentations obtenues sont d'un réalisme flagrant (Figure 15). D'un autre côté, il n'existe aucune possibilité pour permettre son adaptation à une autre application ni son passage à une dimension supérieure. Encore une fois, il s'agit d'un modèle dont le contrôle s'effectue de manière algorithmique mais qui prend en compte des aspects propres au domaine en utilisant des connaissances issues de la botanique.

Les recherches menées par le CIRAD sur l'AMAP ont donné naissance à plusieurs logiciels commercialisés par la société Bionatics™ ([Bionatics 2001]) : natFX™, EASYnat™ (Figure 15), REALnat™, chacun répondant à un besoin particulier et s'adressant à un public déterminé (par exemple natFX™ permet la modélisation de plantes réalistes pour leur incorporation dans les jeux vidéo).

Figure 15 : Paysage modélisé par EasyNat™

3.3.3 *Cartes modulaires et extensions*

Afin de modéliser des feuilles, des fleurs ou même des papillons, on utilise le plus souvent des subdivisions de surfaces. [Eyrolles, et al. 1989] a par exemple recourt à une méthode très simple qui consiste à générer la structure nerveuse de la feuille et à joindre l'extrémité des axes pour définir des surfaces, auxquelles on associe des textures pour un rendu plus naturel (Figure 16). Malheureusement, cette méthode présente quelques inconvénients. Par exemple, il n'est pas rare qu'une nervure n'atteigne pas le bord de la feuille. Il est, de plus, difficile de simuler une surface non plane. Enfin, la coloration est un paramètre complexe qui évolue dans le temps en fonction de critères qui ne sont pas compris par cette méthode.

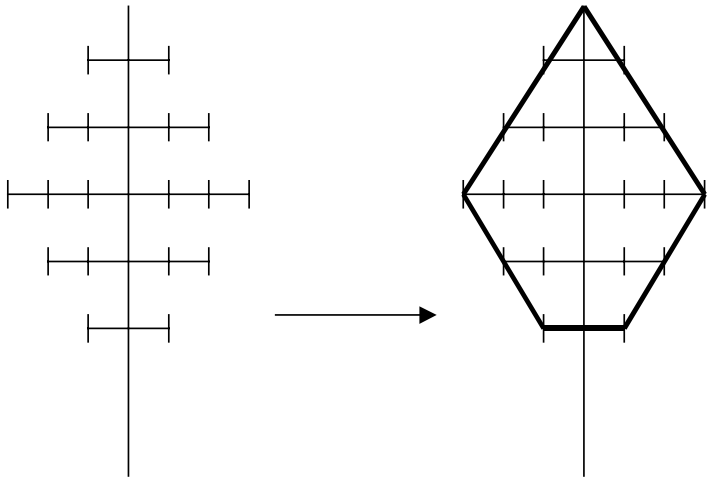


Figure 16 : Illustration de la méthode de [Eyrolles et al. 1989] pour modéliser une feuille surfacique

[Lienhardt 1987] utilise une technique plus complexe mais beaucoup plus puissante basée sur le concept de carte modulaire. Elle repose sur le même principe que celle d'[Eyrolles et al. 1989], à savoir qu'il existe une correspondance entre la forme de la feuille et son réseau de nervures, mais elle y ajoute une notion importante, celle du degré de « collage » entre les différentes parties de l'objet modélisé. Pour cette raison, les cartes modulaires sont structurées comme ceci ([Lienhardt 1987]) :

- certains sommets et arêtes forment un arbre : le C-arbre, correspondant aux nervures de la feuille. Le C-arbre est composé de C-sommets et de C-arêtes. Comme les arbres de l'AMAP, un C-arbre est orienté, et un C-sommet est la racine de l'arbre. Un C-arbre est structuré en axes, en associant un ordre à chaque C-arête, ce qui permet de distinguer les nervures principales, secondaires, etc. ;
- deux faces triangulaires (C-faces) sont incidentes à chaque C-arête, définissant ainsi une petite portion de limbe entourant la nervure. Le bord d'une C-face est composé par une C-arête et ses C-sommets adjacents, ainsi que par deux R-arêtes et un R-sommet (les cellules de la subdivision sont soit des C-cellules soit des R-cellules) ;
- les R-faces « collent » les C-faces entre elles. Leur bord est composé de R-sommets et de R-arêtes. L'évolution du « degré de collage » entre les parties de la surface modélisée est obtenu par opérations sur les R-faces.

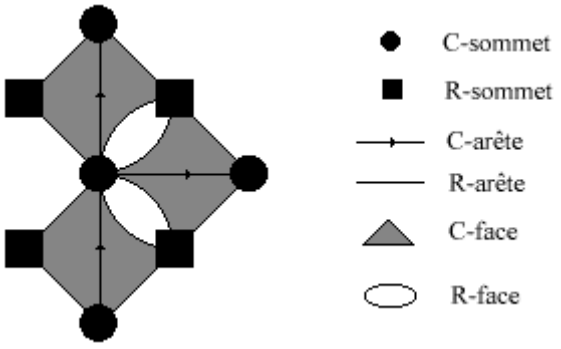
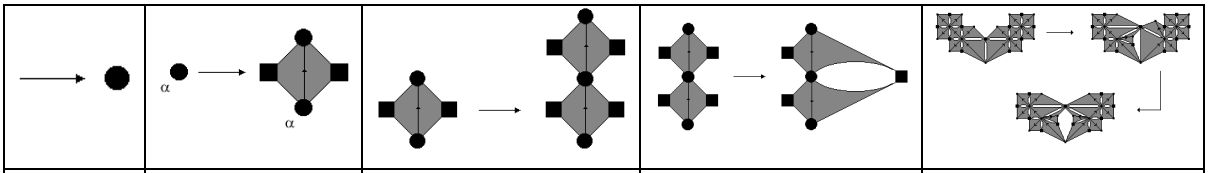


Figure 17 : Représentation graphique des C et R-cellules

Quatre opérations de base sont définies pour manipuler les cartes modulaires (Figure 18). Elles permettent de construire une grande classe de subdivisions de surfaces et de métamorphoses. La première opération crée une carte modulaire initiale, composée d'un C-sommet (racine du C-arbre) et d'une R-face. La deuxième opération ajoute un module dans une carte modulaire, initiale ou non, un module étant principalement composé de 2 C-faces incidentes à une C-arête.

En fait, ces opérations produisent des surfaces « squelettiques », et il est nécessaire d'ajouter des opérations pour manipuler les R-faces. Les deux opérations suivantes sont la création et la modification de R-faces. Elles sont principalement utilisées pour contrôler et simuler des évolutions du « degré de collage » des parties de la surface modélisée.



Création d'une carte initiale	Adjonction d'un module dans une carte initiale	Adjonction d'un module dans une carte non initiale	Création d'une R-Face	Modification d'une R-Face

Figure 18 : opérations sur les Cartes Modulaires

Les comportements sont associés aux C-arêtes, puisqu'elles sont les arguments des principales opérations topologiques. Comme pour d'autres méthodes, les comportements sont des opérations de haut niveau définies comme des séquences d'opérations de base. Ils sont paramétrés, et les paramètres sont similaires à ceux d'autres méthodes déjà étudiées : le temps, la date de création d'une cellule, la distance topologique entre cellules, etc. Certains de ces paramètres (et donc les comportements) sont dépendants du contexte. Ils sont calculés en utilisant les relations d'adjacence et d'incidence entre cellules, comme classiquement en modélisation géométrique.

Par exemple, les distances entre C-arêtes dans le C-arbre permettent de contrôler la topologie, le plongement et la couleur d'une carte modulaire. La topologie d'un C-arbre, et donc la topologie d'une carte modulaire, peut être caractérisée par la longueur des axes, qui peuvent dépendre de la distance entre l'origine de l'axe et la racine du C-arbre. Les couleurs des différentes zones d'une feuille sont souvent fortement liées à la localisation de la zone dans le limbe de la feuille, et cette localisation peut être caractérisée par la distance topologique entre la zone et l'origine de l'arbre des nervures. De la même façon, le « degré de collage » entre lobes peut dépendre de ces distances. De même, on peut faire varier (assombrir par exemple) la couleur selon les distances.

Le plongement des cartes modulaires est principalement défini par le plongement du C-arbre, d'une façon hiérarchique (le plongement d'un C-sommet est fonction du plongement du C-sommet père). En faisant varier les paramètres géométriques, la métamorphose du plongement d'une carte modulaire peut être réalisée, afin de simuler, par exemple le repliement d'une feuille en automne. Le plongement des R-sommets est calculé en fonction du plongement des C-sommets adjacents.

En fait, peu de comportements différents ont été nécessaires pour construire des formes et des métamorphoses complexes. Ceci est dû au fait que les surfaces sont structurées. Par exemple, les fleurs et les feuilles ont des comportements très similaires, qui sont contrôlés par peu de paramètres (les formes naturelles ont souvent des axes de croissance principaux, ce qui se traduit facilement en termes de cartes modulaires).

Cette méthode a été étendue par [Terraz 1994] pour permettre la construction de métamorphoses de tout type de subdivisions de surface, orientable ou non, avec ou sans bords. La topologie de ces subdivisions est modélisée par des 2-G-Cartes. On utilise une structure de subdivisions de base sur laquelle on peut définir des structures de plus haut niveau correspondant à des applications spécifiques. La structure de base est un arbre de faces, lui-même structuré en axes. En fait, les comportements sont associés aux faces sur lesquelles est définie une relation « Père - Fils » (i.e. chaque face est créée en appliquant le scénario d'une autre face). Les faces correspondent aux nœuds de l'arbre, et les relations de filiation aux arêtes de l'arbre (l'arbre étant orienté). Des ordres sont associés à chaque face pour définir les axes de l'arbre. On manipule les subdivisions avec des opérations simples : création de faces, segmentation d'une arête ou d'une face, identification de deux arêtes.

Pour contrôler la métamorphose on utilise des paramètres. Puisque nous avons principalement à contrôler une métamorphose d'arbre, beaucoup de paramètres sont similaires à ceux vus dans les méthodes précédentes.

Beaucoup d'exemples de métamorphoses ont été réalisés avec cette méthode : métamorphoses de surfaces géométriques (dont la surface clef est une surface plane, un cylindre, un tore, etc.), naturelles, ou encore à vocation volumiques (poissons). Une structure de surface de plus haut niveau est définie pour simuler la métamorphose des poissons, puisque les poissons sont également structurés : tête, corps, nageoires, etc.

Comme pour les systèmes de particules, l'utilisation des cartes modulaires présente les avantages et les inconvénients propres aux méthodes générales. Ainsi, il est possible de construire une large classe d'objets et de métamorphoses mais, en contrepartie, aucune connaissance spécifique n'est incluse dans la structure (comme c'est le cas dans AMAP par exemple) pour aider l'utilisateur. Celui-ci doit analyser l'objet ou la métamorphose qu'il désire construire de façon à définir une structure pratique et des comportements complémentaires.

3.4 Conclusion partielle

Depuis le début des années 80 et les premiers travaux sur la simulation de croissance d'objets naturels, de nombreuses méthodes basées sur la topologie ont vu le jour. Elles reposent sur un certain nombre de principes communs : la discrétisation du temps, la décomposition systématique de l'objet en sous-objets ou encore l'association d'un comportement particulier à chaque sous-objet. En plus de ces propriétés propres aux modèles, ces méthodes présentent également la particularité d'être adaptées à un contrôle niveau « animateur », c'est-à-dire par un algorithme.

Ce qui ressort de cette étude est la façon dont chaque méthode exploite la structuration des objets pour associer un comportement aux différents niveaux de structuration. Ainsi, l'animation du vol d'un groupe d'oiseaux par un système de particules est-elle liée au comportement particulier de chaque individu, ainsi qu'aux paramètres globaux de l'ensemble. De même, la croissance d'un arbre modélisée par la méthode de l'AMAP est dirigée par le comportement individuel des bourgeons, mais également par des paramètres globaux comme l'âge de l'arbre.

Les modifications structurelles que permettent de modéliser ces méthodes contraignent le contrôle de l'animation à être de niveau « animateur ». En effet, les paramètres rentrant en jeu, ainsi que les contrôles complexes à effectuer, ne permettent pas de se reposer uniquement, par exemple, sur des lois exprimées sous la forme de fonctions mathématiques.

Tous les systèmes existants (comme TOPL System [Terraz 1994]) reposant sur ces techniques font directement appel à un langage de programmation pour la description d'une évolution. Ainsi, l'utilisateur doit-il être un programmeur pour avoir une chance d'espérer décrire une métamorphose.

Afin de comprendre de manière précise les besoins algorithmiques réels de la simulation d'une métamorphose d'objet naturel, nous avons mené une étude sur un certain nombre d'exemples afin d'en exhiber les concepts. Dans la partie suivante, nous exposons cette étude, laquelle se trouve basée sur une réflexion algorithmique des évolutions analysées.

3.5 Outils algorithmiques pour la construction de métamorphoses

Nous avons vu au cours de l'étude précédente portant sur plusieurs méthodes d'animation que les techniques les plus utilisées pour simuler la croissance d'objets naturels était celles dites « à base topologique ». Ces méthodes présentent la particularité d'être adaptées à un contrôle algorithmique du processus de métamorphose. Ainsi, la plupart des systèmes reposant sur des modèles à base topologique n'offrent ni plus ni moins qu'un langage de programmation et une bibliothèque adaptée pour permettre à un utilisateur de décrire une transformation. C'est le cas par exemple de TOPL System ([Terraz 1994]).

Dans la mesure où nous désirons mettre en place un système d'aide à la programmation de métamorphoses d'objets naturels, il nous a semblé nécessaire d'étudier de près les processus algorithmiques permettant de les décrire. De ce fait, nous avons entrepris une étude de plusieurs exemples de croissance afin de faire ressortir les outils minimaux permettant de faire croître un arbre en 1D½. La 1D½ désigne des objets dont le modèle est en une dimension mais dont la représentation à l'écran est, elle, en deux dimensions.

Notons qu'à ce point, nous laissons volontairement de côté les méthodes basées sur les grammaires car, comme nous l'avons expliqué, celles-ci nous paraissent devenir trop complexes et se rapprocher de manière trop systématique de véritables langages de programmation dès que l'on étudie des comportements non triviaux. L'analyse menée dans cette partie est donc basée uniquement sur un raisonnement algorithmique.

3.5.1 Structure

On étudie ici les développements possibles en nous basant sur une structure en 1D½ correspondant à une structure d'arbre classique (comme celle utilisée dans les méthodes issues de l'AMAP). Un arbre est ainsi constitué de nœuds (ou sommets), et d'entrenœuds (ou arêtes), ceux-ci étant organisés en axes de différents ordres (Figure 19). Le tronc est un axe d'ordre 1, les branches qui sortent de ce tronc des axes d'ordre 2, etc.

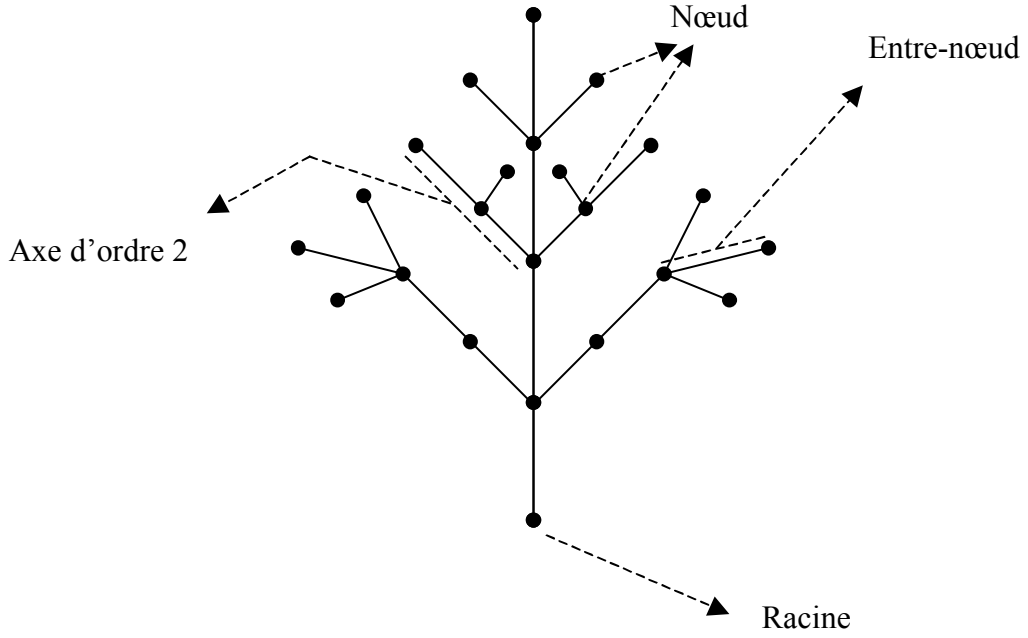


Figure 19 : Structure d'arbre en 1D$^{1/2}$

3.5.2 Modélisation du temps et exécution du programme

Avant d'étudier des exemples précis, il est indispensable de connaître les « habitudes » des programmeurs de métamorphoses. Parmi celles-ci, il faut noter la représentation du temps ainsi que la méthode d'exécution du programme simulant la croissance.

Dans la programmation de croissance selon les modèles à base topologique, le temps est discrétisé. A chaque top d'horloge, on connaît ainsi l'état de l'objet et de ses sous-objets et on applique à ceux-ci les transformations qui leur sont associées. Un programme classique de création d'une métamorphose est donc systématiquement basé sur une répétition implicite du type :

Pour tous les tops d'horloge t **Faire**

{instructions}

Fin Pour

En ce qui concerne le contrôle de l'exécution du programme, celui-ci est géré par la structure même de l'objet. Un concepteur de métamorphose a ainsi pour habitude d'identifier une partie de la structure de l'objet (rappelons-le, décomposé en sous-objets

dans les méthodes à base topologique) et de lui attribuer le parcours de la structure correspondant au déroulement du programme.

Par exemple, dans le cas d'un arbre, chaque axe porte une partie du comportement de l'arbre et, passer d'une étape p à une étape p+1 du développement revient à parcourir l'arbre, axe par axe, et à appliquer le comportement associé à l'axe courant.

Nous aurons l'occasion, dans la suite de ce manuscrit, de revenir sur ce point crucial de la simulation de métamorphoses par une méthode à base topologique. Dans un cadre algorithmique, ce mode de fonctionnement revient à associer un sous-programme à une partie du modèle, et à parcourir le modèle dans un ordre donné, pour exécuter séquentiellement les différents sous-programmes.

3.5.3 Premier exemple

Afin de faire ressortir tous les outils nécessaires à la simulation de croissance d'arbres en 1D½, nous avons donc étudié un certain nombre d'exemples. A partir de quelques métamorphoses, nous avons exhibé des algorithmes correspondant aux développements observés.

Pour des raisons de lisibilité tous les exemples étudiés ne figurent pas ici. Le lecteur est invité à consulter les annexes pour un examen plus approfondi des différents cas. On notera au passage que les exemples utilisés ne sont pas tous des arbres 1D½ mais que, dans le cadre de la présente étude, nous nous sommes limité à l'aspect squelettique des objets, pouvant se ramener systématiquement à la structure présentée au début de cette partie. Notons également que certains exemples sont des feuilles, dont la structure peut se rapporter à celle des arbres. Enfin, certains des exemples présentent des arbres munis de feuilles ou de fleurs : nous nous sommes, dans ce cas, bornés à l'étude des axes, sans prendre en compte ces deux terminaisons.

A titre d'exemple, nous proposons d'observer les transformations subies par une feuille de hêtre au cours de différentes étapes de sa vie (Figure 20).

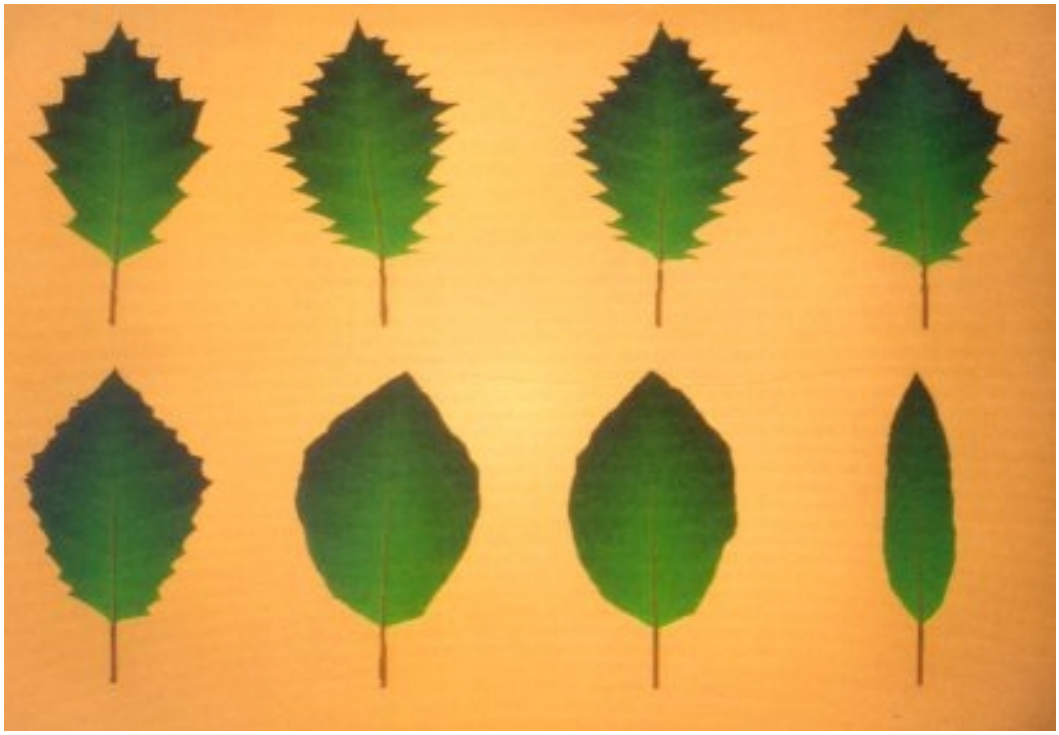


Figure 20 : Exemple de croissance, une feuille de hêtre

3.5.3.1 Description informelle de l'algorithme de croissance

Dans ce paragraphe, nous décrivons de manière informelle la croissance observée sur la Figure 20, en détaillant les événements se produisant sur les axes de différents ordres.

- Axe d'ordre 1 (Tronc) : La croissance de l'axe se déroule selon l'ajout régulier d'arêtes (ou entrenœuds) en son sommet, à une certaine fréquence, et jusqu'à ce que sa taille (en terme de nombre d'entrenœuds) atteigne une certaine valeur (une autre interprétation serait de dire que l'axe croît jusqu'à ce qu'il atteigne un certain âge). Pendant le même temps, les entrenœuds voient leur longueur géométrique augmenter jusqu'à une certaine valeur limite. D'autre part, une insertion d'axes d'ordre 2 a également lieu, à une certaine fréquence.
- Axes d'ordre 2 : Ici, l'ajout d'entrenœuds se déroule à une certaine fréquence jusqu'à ce que la taille (en terme de nombre d'arête) de l'axe soit égale à la distance entre l'origine de l'axe et la racine de l'arbre. La croissance des entrenœuds continue jusqu'à un certain âge (ou une certaine longueur). Les axes d'ordre 3 sont insérés à une date t à partir du ième nœud de l'axe (en partant de la fin). A partir d'une certaine date, les entrenœuds terminaux sont supprimés à une vitesse proportionnelle à la taille de l'axe.
- Axes d'ordre 3 : Les entrenœuds sont ajoutés au sommet de l'axe à une certaine fréquence jusqu'à ce que la taille soit égale à la distance entre l'origine de l'axe et l'extrémité de l'axe support. La croissance des entrenœuds est continue jusqu'à un certain âge (ou une certaine longueur).

Remarques :

- Entre l'étape 3 et l'étape 6, il y a création progressive d'une surface entre les axes d'ordre 2 et les axes d'ordre 3 qu'ils supportent ;
- Les axes d'ordre 3 ne sont supprimés que si l'arête qui les supporte est supprimée.

3.5.3.2 Pseudo algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) Alors
                                Si Date Modulo Frequence1 = 0 Alors
                                        Ajouter_Arete
                                FinSi
                                Si Date Modulo Frequence2 = 0 Alors
                                        Ajouter_Axe (Ordre => 2)
                                FinSi
                        FinSi

                2 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < Age1 Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) = AgeN1) Alors
                                Ajouter_Axe (Ordre => 3, Nœud => i)
                        FinSi
                        Si (Age(AX) <= AgeN2) Alors
                                Si (Taille(AX) < f(AX, Axe (Ordre => 1))) et (Date Modulo Frequence = 0) Alors
                                -- Avec f(AX, Axe(Ordre => 1) = Distance(Origine(AX),Origine(Axe(Ordre => 1)))
                                        Ajouter_Arete
                                FinSi
                        SinonSi (Age(AX) > AgeN2) et (Date Modulo f(Taille(AX))= 0) Alors
                                -- f est une fonction croissante. Plus la taille de l'axe est grande, plus la
                                -- fréquence de suppression des arêtes est élevée.
                                Supprimer_Arete (Arete_terminale(AX))
                        FinSi

                3 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < Age1 Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si Taille (AX) < Distance (Origine (AX), Extremite (Axe_Support(AX))) Alors
                                Ajouter_Arete
                        FinSi

                Autres => Rien

                FinSelon
        FinPour
FinPour
```

3.5.4 Opérations

L'étude de ces différents exemples nous renseigne sur les opérations utilisées, ainsi que sur les structures de contrôles et les paramètres des actions dont le programmeur a besoin pour décrire une métamorphose.

L'évolution d'un objet demande l'intervention de deux types de modifications : les variations géométriques et les variations topologiques.

3.5.4.1 Variations géométriques

Les modifications géométriques affectent les propriétés géométriques de l'objet : en l'occurrence, il s'agit de modifier la longueur des entrenœuds, l'angle d'insertion des axes, etc.

Dans le domaine de la simulation de métamorphose, ces modifications géométriques ont l'habitude d'être exprimées en terme de fonctions d'interpolation, linéaires ou non, simples ou doubles.

[Hachette 2002] donne la définition suivante de l'interpolation :

> ***Interpolation*** *n. fém.* ***MATH****. Méthode permettant de calculer, à partir d'une table de valeurs numériques d'une fonction, les valeurs de cette fonction pour des arguments non présents dans la table. Interpolation linéaire : méthode d'interpolation consistant à figurer par une droite les variations de la fonction entre deux arguments consécutifs.*

L'interpolation est donc une prolongation par continuité (de façon linéaire ou sinusoïdale). C'est une loi d'évolution entre deux valeurs cartésiennes. En règle générale, on se donne deux points et une loi d'évolution, et, à partir de ces informations, on déduit toutes les valeurs intermédiaires.

Les programmeurs de métamorphoses utilisent des interpolations simples et des interpolations doubles.

Interpolation simple.

Si on connaît deux valeurs la, lb à deux instants a et b donnés, on peut en déduire toutes les valeurs dans l'intervalle de temps [a, b] (Figure 21).

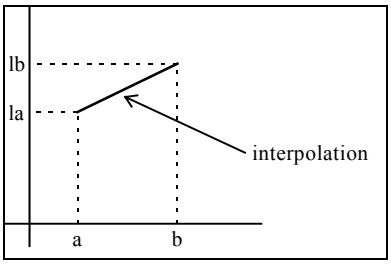


Figure 21 : Représentation graphique d'une interpolation linéaire [a, b] → [la, lb]

Les interpolations simples sont utilisées pour décrire des évolutions simples de longueur ou d'angle. Typiquement, on peut déduire des longueurs l_1 et l_2 d'un entrenœud à deux

étapes différentes e_1 et e^2, la longueur de cet entrenœud pour toutes les étapes de l'intervalle [e_1, e_2] (Figure 22).

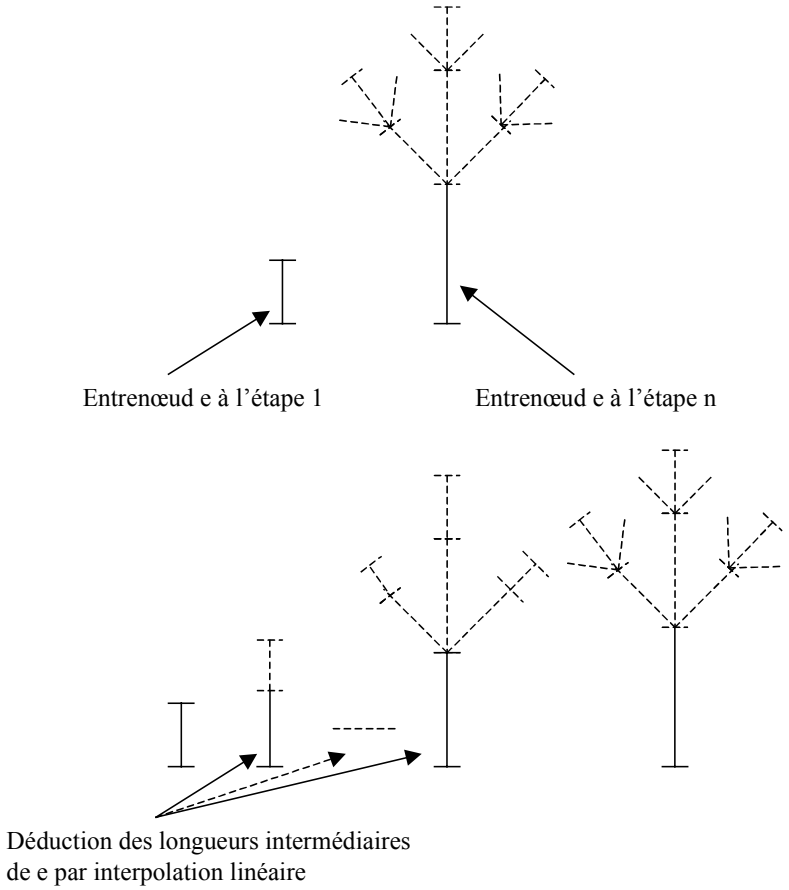


Figure 22 : Utilisation de l'interpolation simple pour déterminer la longueur d'un entrenoeud à différentes étapes de la maturation de l'arbre

Interpolation double.

Il s'agit d'une interpolation effectuée sur le résultat d'une première interpolation.

Prenons un exemple : on dispose d'un descriptif nous donnant la longueur des entrenœuds d'un axe à sa maturité. Généralement c'est une interpolation simple en fonction de la distance (en terme d'entrenœuds) séparant un entrenœud de l'origine de l'axe (Figure 23).

Interpolation [Dmin, Dmax] -> [Lmin, Lmax]

où Lmin et Lmax sont les longueurs des entrenœuds aux distances Dmax et Dmin.

Figure 23 : Un axe à une étape donnée de sa maturation. La longueur des entrenœuds est le résultat d'une interpolation linéaire simple, fonction de la distance des entrenœuds à l'origine de l'axe.

Cela signifie que grâce à l'interpolation linéaire en fonction de la distance, on peut connaître la longueur de tout entrenœud de l'axe à sa maturité simplement avec sa distance à l'origine de l'axe

Si d'autre part on considère que la longueur d'un entrenœud varie entre 0 pour sa date de création et la longueur Max pour sa date de maturité (longueur donnée par l'interpolation précédente), on peut alors calculer sa longueur à une étape de la croissance donnée par une interpolation en fonction de son âge (Figure 24).

```
Interpolation [Age0, AgeMax] -> [Lmin, Lmax]
```

où Lmin et Lmax sont cette fois les longueurs des entrenœuds aux dates Age0 et AgeMax.

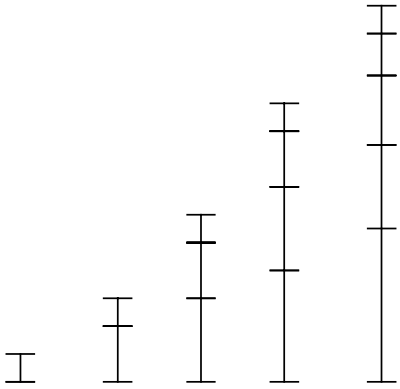
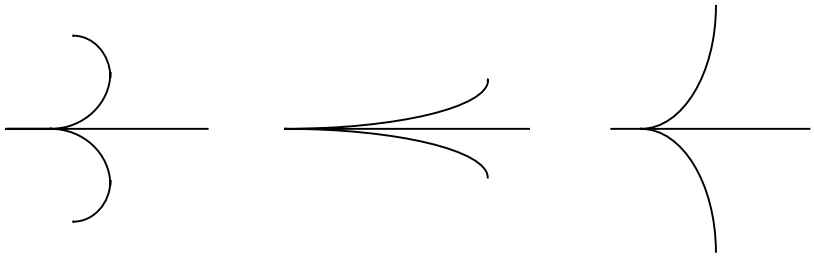


Figure 24 : Interpolation réalisée sur le résultat d'une première interpolation

On parle dans un exemple comme celui-ci d'interpolations en espace et en temps. L'interpolation en espace permet de connaître la longueur des différents entrenœuds d'un axe à une étape donnée, l'interpolation en temps permet de déduire les longueurs d'un entrenœud aux différentes étapes de sa maturation.

L'***interpolation*** est l'opération de base utilisée pour modifier la géométrie de la structure. En l'occurrence, elle permet de calculer la variation des longueurs et des angles des arêtes de l'arbre. Tous les exemples ci-dessus portent sur des interpolations de longueur mais,

dans les exemples que nous avons étudié, l'opération d'interpolation s'utilise également pour calculer les angles des entrenœuds, comme c'est le cas sur la Figure 25.



```
Pour tous les tops d'horloge Faire
        -- Pour chaque arête : Angle formé en fonction de la distance à l'origine de l'axe à t = Textremite

        Interpolation [DistanceMin, DistanceMax] -> [AngleDMin, AngleDMax] (à t = Textrémité2)

        -- Pour chaque arête : Angle formé en fonction de son âge (connaissant les angles min et max au cours de
        -- son évolution).

        Interpolation [AgeExtremite1, AgeExtremité2] -> [-AngleMax, AngleMax]

        -- Il faut prolonger ce résultat à tous les tops d'horloge : tout ce qui est vrai à
        -- un instant t, l'est également à t modulo (AfeExtremite2 - AgeExtremite1) et à
        -- (-t) modulo (AgeExtremite2 - AgeExtremite1) :
        -- Interpolation [AgeExtremite1 modulo (NTO), AgeExtremite2 modulo (NTO)] -> [-AngleMax, AngleMax]
        -- Interpolation [-AgeExtremite2 modulo (NTO), -AgeExtremite1 modulo (NTO)] -> [-AngleMax, AngleMax]
        -- Où NTO est le nombre de tops d'horloge durant une oscillation = AgeExtremite2 - AgeExtremite1
FinPour
```

Figure 25 : Oscillations d'un axe dont la description passe par une interpolation de l'angle

3.5.4.2 Variations topologiques

Les variations topologiques sont les modifications effectuées sur la structure de l'objet lui-même : en ce qui nous concerne, sur les exemples étudiés, elles se cantonnent à la création d'un arbre, à l'ajout et à la suppression d'arêtes et d'axes.

On distingue par la suite la croissance de l'axe proprement dite, c'est-à-dire l

3.5.4.2.1 Croissance d'un axe

Dans le cas de la variation sur un axe unique, on utilise l'unique opération de base : « ajout d'une arête à l'extrémité de l'axe ».

La question que l'on se pose est d'ordre temporelle (puisqu'on sait déjà où va se produire la modification) : « QUAND doit-on ajouter une nouvelle arête ? ». Il s'agit donc ici de contrôler le rythme de croissance de l'axe. Pour ça on utilise la fonction ***modulo***. Par exemple :

- Création d'arêtes tous les tops d'horloge modulo n ;
- Création d'arêtes tous les tops d'horloge modulo (âge de la dernière arête) ;
- Etc.

Le rythme peut être constant, évoluer au cours du temps (accélération, décélération) ou encore dépendre de certaines conditions. Dans ce dernier cas, la condition de croissance

est combinée avec des opérations logiques et la structure conditionnelle classique « Si-Alors-Sinon ». Ainsi, les conditions peuvent porter sur la taille de l'axe (en terme de nombre d'arêtes), sur la longueur de l'axe (en terme de somme des longueurs des différentes arêtes qui le composent), sur le temps (pendant une certaine période) ; etc.

De même, l'arrêt de la croissance peut être conditionné par la taille de l'axe (on stoppe lorsque l'axe atteint une certaine taille) ou par le temps (on stoppe la croissance à une certaine date).

3.5.4.2.2 Insertion d'axes secondaires

L'insertion d'axes secondaires ajoute une difficulté supplémentaire. En plus de la question sur le contrôle du rythme (niveau temporel), il faut gérer la position d'insertion des axes secondaires (niveau spatial). Ainsi, quand on insère un axe secondaire sur un axe donné, on doit non seulement se demander à quel moment mais également où, en terme de positionnement sur l'axe porteur, on doit l'insérer.

Pour répondre à ces deux questions on doit utiliser, comme précédemment, la fonction modulo, chargée de régenter le rythme de croissance, et on doit lui adjoindre une méthode permettant de connaître la position d'un entrenœud dans l'axe, par rapport à l'origine de cet axe. On contrôle ainsi le positionnement de l'insertion des axes secondaires par des instructions comme :

- Insérer un axe secondaire tous les deux tops d'horloge, au niveau de l'entrenœud numéro (longueur de l'axe – 1) ;
- Insérer un axe secondaire tous les tops d'horloge, au niveau du dernier entrenœud de l'axe ;
- Etc.

Les variations de rythme d'insertions sont gérées par des contrôles du même type que ceux régulant la croissance d'un axe : le rythme peut être constant, évoluer au cours du temps (accélération, décélération) ou encore dépendre de certaines conditions (utilisation d'opérations logiques et de structures conditionnelles).

3.5.5 Récapitulatif

Ce paragraphe propose un récapitulatif de tous les outils recensés lors de notre étude de cas.

Outils mathématiques

- Fonctions d'interpolation ;
- Fonction Modulo ;

Opérations constructives

- Création d'un arbre ;
- Insertion d'un axe secondaire ;
- Ajout d'un entrenœud au sommet d'un axe ;

Opérations de modifications géométriques

- Modification de l'angle d'un entrenœud ;
- Modification de la longueur d'un entrenœud ;

Opérations de calculs

- Taille d'un axe (en terme d'entrenœuds) ;
- Longueur d'un axe (en terme de somme des longueurs géométriques des entrenœuds) ;
- Ordre d'un axe ;
- Age d'un axe ;
- Longueur d'un entrenœud ;
- Angle d'un entrenœud ;
- Age d'un entrenœud ;
- Position d'un entrenœud dans un axe ;

Structures de contrôle

- Conditions ;
- Boucles ;

Algorithmes

- Parcours de la structure ;

3.5.6 Conclusion

Cette étude de plusieurs exemples de développements d'arbres en 1D½ nous permet de cerner une manière particulière de description d'une métamorphose. En supposant que les objets étaient structurés et modélisés par une méthode à base topologique, nous avons montré combien la description d'une métamorphose se prêtait particulièrement bien à un cadre algorithmique : utilisation de variables, de structures de contrôles, etc. Ce que nous montrent en outre ces exemples, c'est que comprendre le scénario de croissance d'un objet revient à interpréter un langage de programmation fonctionnelle. Il faut en effet être capable de reconnaître les commandes, les paramètres, etc.

4 Conclusion

Nous avons étudié dans cette partie les différentes approches existant pour modéliser des animations et représenter des métamorphoses d'objets naturels. Le problème majeur que soulève cette étude est un problème classique de la programmation : il est difficile de concilier une description de haut niveau, facilement utilisable, à un contrôle précis, nécessitant en conséquence un effort plus important de la part de l'utilisateur. Les applications de l'animation n'étant pas systématiquement destinées à des informaticiens, les méthodes les plus précises et les plus proches de la programmation ont été logiquement délaissées au profit de méthodes de plus haut niveau mais offrant une précision de description

moindre. Pour les mêmes raisons, les méthodes permettant de modifier la structure des objets, comme les méthodes à base topologique, sont écartées car inutilisables par des experts du domaine non programmeurs.

En dehors des approches par grammaires, dont le contrôle prend de plus en plus la forme d'un programme informatique, la description de simulation de croissance d'objets naturels est un domaine réservé aux méthodes à base topologique, qui elles seules permettent une modification de la structure des objets. Malheureusement, ces méthodes sont contrôlées systématiquement de manière algorithmique. Pour cette raison, alors qu'il existe de nombreux logiciels commerciaux permettant à un simple utilisateur de créer l'animation d'un personnage ou de simuler la déformation d'un objet de manière intuitive, aucun système ne permet de décrire de manière interactive les modifications structurelles nécessaires à la modélisation d'une métamorphose. Par exemple, XFrog™ ([Lintermann & Deussen 1999], Figure 26) est un des logiciels les plus aboutis dans la création d'objets naturels pour des non programmeurs. Il propose, à partir de différents éléments prédéfinis, de composer interactivement un objet en le décrivant par un diagramme selon ses ramifications, ses terminaisons, etc. Le système propose même d'effectuer des animations en utilisant des interpolations pour modifier la géométrie des objets créés. Malheureusement, il n'est pas possible d'en spécifier les évolutions topologiques au cours du temps.

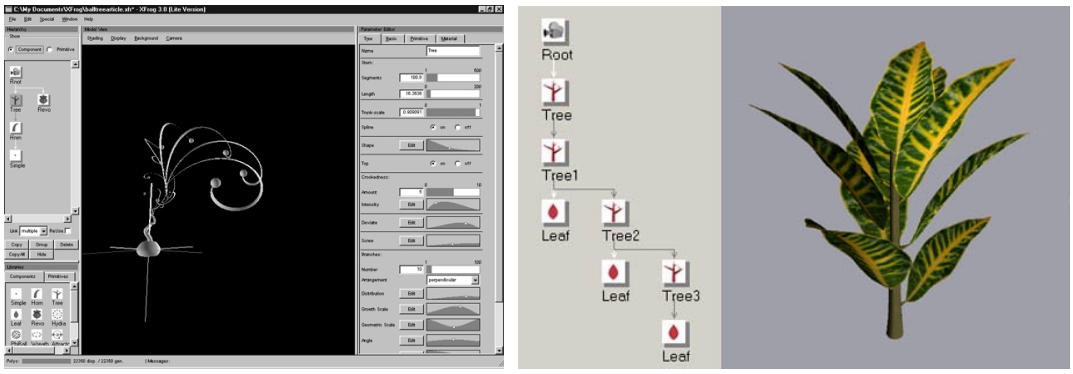


Figure 26 : Xfrog, un logiciel de modélisation d'objets structurés pour non programmeurs

Les seuls outils permettant de modifier la structure des objets modélisés proposent des environnements textuels, véritable environnements de programmation, et, dans le meilleur des cas, les agrémentent de panneaux permettant de modifier les valeurs de quelques paramètres prédéfinis (TOPL System dans [Terraz 1994]).

Pour décrire de telles transformations, la technique usuelle consiste donc à prendre son langage de programmation préféré et à s'attacher, via un bon éditeur de texte, à coder pas à pas toute la métamorphose désirée. Dans un deuxième temps, le concepteur doit compiler son programme et l'exécuter pour en tester la validité. Enfin, il doit corriger les inévitables erreurs et recommencer un cycle de prototypage type : écriture - compilation - test.

Cette approche pose au moins deux problèmes majeurs. Tout d'abord, il y a un décalage permanent entre la représentation textuelle du programme et son exécution graphique. Dans un domaine dans lequel la notion de prototypage est aussi déterminante, il semble-

rait naturel que l'on puisse passer instantanément du mode de conception au mode d'exécution sans avoir à projeter mentalement une représentation à l'intérieur d'une autre. D'autre part, pour un public souvent constitué de non-informaticiens et en tout cas de non-programmeurs, le mur représenté par la pratique d'un langage de programmation constitue simplement un obstacle infranchissable.

Pour offrir à des concepteurs non programmeurs un environnement de création de métamorphoses plus convivial, nous devons donc utiliser des techniques interactives de génération de programmes. Dans la partie suivante, nous présentons les outils de conception nécessaires à la mise en place d'un tel environnement.

Chapitre 2

Outils pour la conception de systèmes de C.A.O. de métamorphoses

Résumé. *Tous les modèles d'architectures développés au cours des deux dernières décennies tendent vers le même objectif : permettre au concepteur d'applications interactives de séparer en différents composants la partie propre au domaine d'application, l'interface avec l'utilisateur, et le contrôle de l'application. Des différentes approches envisagées émergent les solutions dites « mixtes », tentant de concilier une description globale de ces composants et des liens qui les unissent à une description précise de leur fonctionnement interne. En s'appuyant sur cette analyse, nous détaillons l'architecture H4, qui, en plus de prendre en compte ces différents aspects, permet à une application interactive de disposer naturellement du dialogue structuré. Ce concept, utilisé principalement en Conception Assistée par Ordinateur pour permettre à un utilisateur de décomposer aisément une tâche complexe en un ensemble de sous-tâches, est adapté à toutes les applications de conception technique. En particulier, dans le cadre d'une application d'aide à la définition de métamorphoses, il est souhaitable que l'utilisateur puisse déterminer une construction de la manière la plus précise possible.*
Entre programmation interactive, programmation visuelle, programmation sur exemple, macros, etc., quelle est la solution la plus adaptée à la programmation d'une métamorphose de manière conviviale ? L'analyse de plusieurs classifications des approches de conception interactive permet de comprendre que la programmation sur exemple est celle qui correspond le mieux à nos attentes et que des solutions existent pour résoudre les problèmes de la nomination automatique des objets, de l'introduction des structures de contrôle ou de la représentation du programme généré.

1 Introduction

Dans le chapitre précédent, nous avons étudié les différentes méthodes permettant de modéliser des animations par ordinateur. Nous avons ainsi pu constater que les méthodes basées sur la topologie prenaient en compte les propriétés structurelles des objets naturels et étaient, de ce fait, les plus adaptées à la simulation de métamorphoses. D'autre part, nous avons également remarqué que les animations de ce type se prêtaient particulière-

ment bien à un contrôle niveau « animateur » : l'association de scénarii aux différents éléments composant un objet, ainsi que le parcours de la structure pour l'exécution du comportement global ou la prise en compte de données extérieurs sous forme de paramètres sont autant de facteurs facilement descriptibles de manière algorithmique.

Le principal écueil à ces méthodes vient justement du caractère fortement algorithmique de la description de la métamorphose. Ainsi, il n'existe aucun système interactif abouti permettant à un utilisateur de détailler les modifications structurelles d'une telle simulation. Les seules solutions existantes consistent à utiliser un véritable langage de programmation, ce qui ne constitue évidemment pas une réponse satisfaisante puisqu'elle restreint le spectre des utilisateurs potentiels à des programmeurs et contraint le concepteur à de longues phases de codage, compilation et test.

Pour cette raison, nous voulons mettre au point un environnement capable de proposer au concepteur d'une application destinée à simuler des métamorphoses, un ensemble d'outils permettant aux futurs utilisateurs de réaliser plus facilement les simulations désirées.

Dans cette partie, nous étudions un certain nombre d'outils, architecturaux et de conception, dans l'optique de mettre en place un tel environnement. Dans un premier temps, nous analysons les besoins d'une application de création de métamorphoses au niveau du dialogue et nous mettons en évidence les faiblesses et les forces de différents systèmes d'architecture par rapport à ces besoins. Dans un deuxième temps, nous passons en revue différentes techniques de création de programme par des utilisateurs non programmeurs. Pour chacune d'elle nous examinons les possibilités présentées au niveau interactif ainsi qu'au niveau des concepts de programmation manipulés.

2 H⁴ : une architecture logicielle pour le dialogue structuré

2.1 Le dialogue structuré

Comme nous l'avons vu dans le chapitre précédent, la modélisation de phénomènes naturels est un domaine réservé à des experts. A l'instar de la CAO ou d'autres domaines très spécialisés, un système interactif de modélisation de phénomènes naturels rentre donc dans le cadre de ce que l'on appelle les Applications Graphiques Interactives de Conception Technique (AGICT). Ces applications proposent de mettre au service d'un expert d'un domaine les outils nécessaires pour que celui-ci soit en mesure de réaliser, de la manière la plus précise possible, toutes les opérations désirées pour atteindre un objectif donné. Laurent Guittet [Guittet 1995] définit une AGICT en ces termes :

> ***AGICT.*** *Une AGICT est une application permettant de créer une représentation informatique d'un objet technique susceptible d'être manipulée par des fonctions programmées de simulation ou de production.*

Au contraire d'applications « grand public », comme Paint™ ou MacDraw™, une AGICT ne se contente pas d'une description approximative comme « dessiner une droite qui passe vers le centre de ce cercle », mais au contraire de contraintes précises, spécifiques au domaine et indispensables au concepteur. Par exemple, un système de CAO doit pouvoir proposer la possibilité de construire un cercle tangent à trois droites, un système de modélisation de métamorphose d'insérer un entrenœud dont la longueur vaut la moitié de l'entrenœud précédent, etc. De la même façon, un système permettant de décrire des croissances de plantes doit autoriser des constructions du type : « Ajouter un entrenœud de longueur égale à la longueur du précédent entrenœud de l'axe AX au sommet de l'axe AX ».

2.1.1 Classification des applications interactives

Pour bien savoir dans quel cadre se situent les applications de conception de métamorphoses qui nous intéressent, nous nous référons à une classification que nous détaillons dans cette section.

La taxonomie des systèmes interactifs établie dans [Pierra 1995] met en évidence les points particuliers d'un domaine d'application. Nous nous basons sur cette étude pour différencier quatre critères qui nous semblent pertinents dans le cadre de nos travaux. Dans ce récapitulatif, nous utilisons le terme **tâche** en qualifiant ainsi un but utilisateur associé à un ensemble de fonctions du système.

2.1.1.1 Arité des tâches

On distingue deux types de tâches.

On dit d'une application qu'elle est **mono-objet** si les tâches qu'elle utilise ne requièrent qu'un seul objet du modèle pour s'exécuter. Par exemple, les applications Paint ou MacDraw sont mono-objet : toutes les actions possibles portent sur un seul objet du modèle (dessiner un cercle, changer la couleur d'un rectangle, etc.).

Les applications mono-objet s'opposent naturellement aux applications dites **multi-objet**. Celles-ci comportent des tâches qui peuvent utiliser plusieurs objets du système pour s'exécuter. Les applications de CAO sont multi-objet. Par exemple, « construire un cercle tangent à trois droites » est une tâche multi-objet puisqu'elle met en jeu trois objets du modèle. Comme l'explique l'auteur dans [Martin 1995], les utilisateurs de systèmes de CAO sont habitués à utiliser des relations sous forme de contraintes ou d'opérateurs géométriques. Dans un registre comparable, un concepteur de métamorphoses utilisant un système interactif veut également pouvoir exprimer des contraintes précises du même type que celles décrites par un utilisateur de système de CAO. « Insérer un axe secondaire sur un axe donné », « insérer un entrenœud à l'extrémité d'un axe de longueur égale à la moitié du dernier entrenœud de cet axe » sont autant de tâches multi-objet que l'on veut être capable de décrire dans un système interactif de modélisation de phénomènes naturels.

Les applications multi-objet utilisent systématiquement un dialogue « préfixé » dans lequel on désigne d'abord la tâche à effectuer puis seulement les objets, paramètres de cette tâche. Nous revenons plus en détail par la suite sur les principales stratégies de dialogue utilisées en conception technique.

2.1.1.2 Structure des tâches

Une tâche est dite **atomique** si elle est indépendante de l'exécution de toute autre tâche. Le résultat produit est enregistré par l'objet du domaine. Par opposition, une tâche **structurée** peut utiliser le résultat d'une autre tâche comme paramètre. Le résultat de son exécution n'est connu que lorsque l'utilisateur a donné la hiérarchie complète tâche / sous-tâche qui la compose. Cette décomposition hiérarchique correspond à l'analyse donnée par Norman [Norman 1986], expliquant qu'un utilisateur est plus apte à résoudre un problème quand il l'a dissocié en une multitude de sous-buts plus accessibles.

Une AGICT est caractérisée par le fait qu'elle est capable de supporter des tâches structurées. Les différents éléments du modèle sont en effet liés par plusieurs types de relations : des relations numériques (la longueur de cet entrenœud vaut trois fois la longueur de celui-ci), géométriques (cet entrenœud s'insère à l'extrémité de cet axe) ou graphonumériques (l'angle d'insertion de l'entrenœud à créer vaut la moitié de la somme des angles des entrenœuds de l'axe). Celles-ci sont connues de l'utilisateur du système et elles doivent être exploitables durant la phase de conception.

De ce fait, les tâches doivent être structurées et une tâche doit pouvoir utiliser le résultat d'une autre tâche comme paramètre (le résultat du calcul de la somme des angles des entrenœuds d'un axe pour créer un entrenœud). Nous nous plaçons dans un cadre où l'on ne peut pas se contenter de dire « je crée un entrenœud et je m'arrange pour que son angle d'insertion vaille grosso modo la moitié de la somme des angles des entrenœuds de l'axe », mais où l'on veut être capable de faire appel à des contraintes de construction précises, tout en conservant un mode d'expression simple.

2.1.1.3 Structure des objets

Les objets du domaine de conception peuvent être considérés comme **structurés** lorsque plusieurs niveaux sont accessibles à l'utilisateur. Au contraire, les objets sont dits **simples** s'ils ne peuvent pas être constitués d'autres objets du modèle. Dans le second cas, la désignation et son écho peuvent être réalisés par la couche de Présentation, chargée des interactions directes avec l'utilisateur. Par contre, dans le cas d'objets structurés, cette interprétation ne peut être se faire que par le composant Domaine, porteur de la sémantique de l'application.

Dans les AGICT, les objets sont fortement structurés. Ainsi, pour un système de CAO, un cube est constitué de faces, elles-mêmes organisées en arêtes, etc. De même, dans le domaine de la métamorphose, les méthodes basées sur la topologie utilisent des modèles structurés : les arbres de l'AMAP sont ainsi formés d'axes, eux-mêmes composés d'entrenœuds, etc. Ainsi, lorsque l'utilisateur du système désigne une entité, le système

doit être capable, selon les besoins, de reconnaître s'il s'agit de l'arbre entier, de l'axe, de l'entrenœud, etc. (Figure 27).

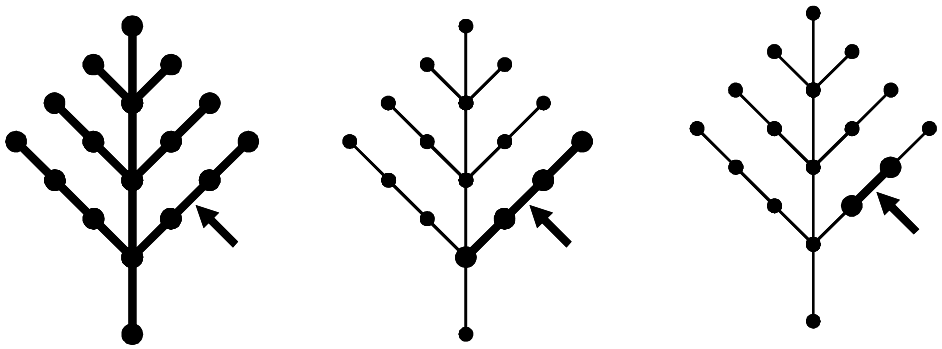


Figure 27: Désignation d'un objet structuré

2.1.1.4 Relation entre objets

Les objets du domaine sont dits **relationnels** si la représentation d'un objet dépend d'autres objets du domaine. Dans le cas contraire, on dit qu'ils sont **autonomes**. Dans le cas où les objets sont liés les uns aux autres, la modification de l'un peut entraîner la modification des autres. Par contre, si un objet est autonome, il peut être mis en bijection avec un objet unique d'interaction chargé entre autre de sa représentation [Duke & Harrison 1993].

D'après cette définition, les entités manipulées dans une AGICT comme un système de CAO ou une application de modélisation de métamorphoses sont relationnelles. En effet, les objets sont fortement liés entre eux : ainsi, par exemple, la visualisation du tronc d'un arbre va-t-elle dépendre des éventuelles branches et feuilles placées devant.

2.1.1.5 Conclusion

A la lecture de ces différents critères de classification, on s'aperçoit qu'une application permettant de décrire la métamorphose d'un objet naturel structuré est comparable à une application de CAO. Il s'agit de systèmes multi-objet, dont les tâches sont structurées et dont les objets sont composés et relationnels.

Forts de cette constatation, nous analysons dans la section suivante le dialogue mis en place dans des applications de ce type.

2.1.2 Stratégies de dialogue en conception technique

La nature structurée des tâches implique un mode de dialogue particulier. On distingue deux types de langages, correspondant à deux méthodes différentes pour décrire une tâche.

Dans un système utilisant un dialogue basé sur un langage **post-fixé**, l'utilisateur doit d'abord fournir l'objet sur lequel porte la tâche avant de désigner l'action elle-même. Par exemple, il va sélectionner un objet puis activer une commande de changement de cou-

leur qui va agir sur cet objet. Ce type de dialogue, utilisé notamment en **Manipulation Directe**, se prête uniquement aux systèmes supportant des tâches atomiques mono-objet. En effet, une fois l'objet de l'action sélectionné, il n'est pas concevable d'en sélectionner un deuxième dans le but de lui fournir le résultat d'une opération ayant eu lieu sur le premier. Un tel dialogue n'est donc pas envisageable dans le cadre d'un système supportant des tâches structurées.

A l'inverse, les langages **préfixés** reposent sur une logique « commande - opérande ». Dans un système basé sur un tel langage, l'utilisateur précise d'abord l'opération qu'il veut effectuer (par exemple en cliquant sur un bouton associé) puis, seulement, il fournit les objets, paramètres de la tâche. D'après [Guittet 1995], il s'agit là du seul type de dialogue utilisable lorsque les tâches d'un système sont structurées. En effet, dans ce mécanisme, un opérande peut très bien être remplacé par une tâche capable de produire l'information attendue.

Si le dialogue préfixé se prête naturellement plus aux AGICT, il ne facilite par contre pas du tout la Manipulation Directe.

2.1.3 Dialogue structuré

> *"Breaking down goals into sub-goals is the natural way for users to solve problems. This decomposition is done recursively until actions of the system may be reached." (Le moyen le plus naturel pour un utilisateur de résoudre un problème est décomposer son but en un ensemble de buts intermédiaires. Cette décomposition se fait de manière récursive jusqu'à ce que l'on atteigne les actions du système) [Norman 1986].*

Le Dialogue Structuré se base sur l'analyse de Norman et met en pratique la décomposition des buts de l'utilisateur en un arbre de buts/sous-buts. Ainsi, mettre en place un tel dialogue revient à fournir à l'utilisateur le moyen de calquer son arbre des tâches sur celui des buts/sous-buts. Les paramètres des tâches sont ainsi exprimés au moyen d'autres tâches et une tâche peut être exécutée dans le contexte d'une autre tâche. Ce dernier point met en exergue la nécessité d'organiser les tâches de manière hiérarchique et de différencier les **tâches terminales** (ou de consommation), des **tâches de production**.

Considérons à titre d'exemple la tâche « créer un segment dont une extrémité est le centre d'un cercle et l'autre l'intersection de deux segments ». Elle peut être décomposée en « récupérer le centre d'un cercle », « récupérer l'intersection de deux segments » et « créer un cercle par deux positions ». Les deux premières tâches fournissent des informations qui peuvent être utilisées par cette dernière. On dit qu'il s'agit de tâches d'un niveau d'abstraction moins élevé (Figure 28). De la même manière, la tâche « Ajouter un entrenœud dont la longueur est égale à la moitié du dernier entrenœud de l'axe AX au sommet de AX », peut se décomposer en « récupérer le dernier entrenœud de AX », « récupérer la longueur du dernier entrenœud de AX », « diviser par deux la longueur du dernier entrenœud de AX », « ajouter un entrenœud à AX dont la longueur vaut la moitié du dernier entrenœud de AX » (Figure 29).

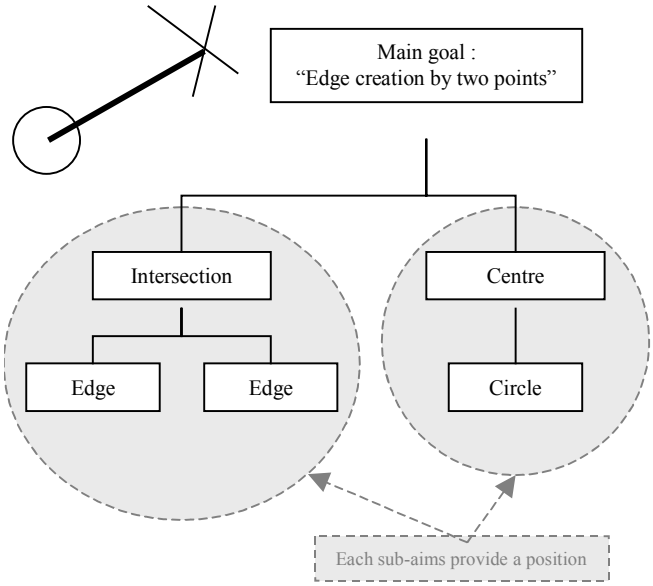


Figure 28 : Création d'un segment par deux points

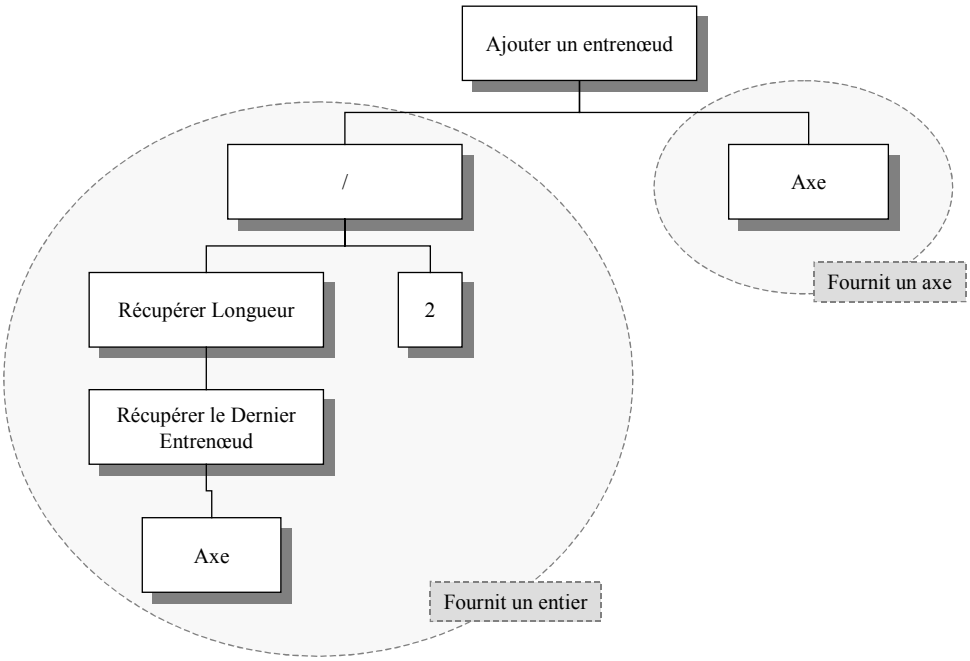


Figure 29 : Création d'un entrenoeud

La mise en œuvre de cette méthode nécessite une architecture particulière ([Guittet 1995], [Gardan, et al. 1988], [Gardan, Jung, & Martin 1993], [Martin 1995]). Il faut en effet que le Contrôleur de Dialogue de l'application soit en mesure de distinguer les tâches terminales des sous-tâches d'expression. Pendant que les premières sont chargées de modifier le modèle ainsi que de prendre en compte certains aspects de la présentation (comme les facteurs de zoom), les secondes mettent le résultat qu'elles calculent (numérique ou géométrique) à la disposition d'autres tâches (ou sous-tâches).

Dans la suite de ce chapitre, nous passons en revue quelques unes des architectures logicielles les plus connues en nous replaçant à chaque fois dans le cadre du dialogue structu-

ré. Nous montrons ainsi comment l'architecture H4, créée spécifiquement pour répondre à ce besoin, permet de prendre en compte une telle logique de dialogue.

2.2 Les modèles d'architecture

Dans le processus de création d'une application, la partie réservée à la programmation proprement dite est généralement la plus facile. La plus grande difficulté consiste à essayer d'appréhender le problème de manière à savoir « quoi » programmer exactement. Tout le temps passé à analyser la façon dont le code doit être organisé constitue par la suite autant d'efforts en moins pour effectivement écrire le programme. Malheureusement, la plupart des gens ne raisonnent pas de cette façon. Pourtant, les nombreuses architectures logicielles sont autant d'aides précieuses à l'élaboration d'une application : ce sont de véritables cadres de pensées qui aident le concepteur à formuler ses idées dans un certain sens puis, naturellement, à décomposer son problème en différents objets et/ou modules.

Un aspect primordial de ces architectures est la description de la décomposition d'une application interactive en différents éléments directeurs. Une telle décomposition permet une conception plus simple dans la mesure où chaque module peut être réalisé de manière plus ou moins indépendante. D'autre part, les coûts d'éventuelles modifications s'en trouvent amoindris et leur fiabilité accrue. Malheureusement, cette décomposition est difficile à obtenir car l'interface se situe à la jonction de l'application et de l'utilisateur. De ce fait, le processus de mise au point est un mécanisme purement itératif [Tarby 1993].

Malgré la grande diversité des modèles d'architecture existants, tous reposent sur un principe commun : la séparation entre le Noyau Fonctionnel, représentant les objets et les méthodes de l'application, et l'Interface, en charge de la présentation à l'utilisateur des objets de cette application. D'autre part, une architecture logicielle décrit également de manière systématique, avec plus ou moins de précision, la manière dont ces deux composants essentiels communiquent entre eux.

Dans cette partie, nous présentons quelques uns des modèles d'architecture les plus utilisés en montrant, pour chacun d'eux, les limites qu'ils présentent vis-à-vis du dialogue structuré. Nous commençons cette description avec les modèles dits « globaux » puis nous montrons comment les progrès techniques en matière de programmation ont amené les concepteurs d'architectures logicielles à se pencher sur des concepts « multi-agents » se rapprochant de l'approche orientée-objet. Enfin, nous décrivons quelques modèles « hybrides » qui tentent de tirer parti des avantages des deux approches précédentes.

2.2.1 Les modèles centralisés

Les modèles centralisés sont historiquement les premiers modèles d'architecture à avoir vu le jour. Ils tentent de répondre à une question primordiale : l'isolement de la partie interface du reste de l'application, afin de rendre cette dernière indépendante de tout matériel ou de tout outil spécifique à la présentation.

2.2.1.1 Le modèle de SEEHEIM

Le modèle de SEEHEIM [Pfaff 1985] est né d'une réunion de travail ayant eu lieu dans la ville de SEEHEIM en 1983. Ce modèle préconise la décomposition d'une application interactive en trois modules distincts, permettant de séparer la partie Interface de la sémantique même de l'application.

La **Présentation** est chargée d'afficher les informations de l'application à l'utilisateur (via l'écran) et de récupérer ses « entrées » (c'est-à-dire toutes les saisies faites à partir des dispositifs mis à sa disposition comme le clavier, la souris, etc.). C'est ici que doivent être prises en compte d'éventuelles règles d'ergonomie. Il est important de noter que cette couche logicielle est indépendante du reste de l'application. Elle n'a aucune connaissance de la sémantique des actions effectuées par l'utilisateur et ne connaît que le contrôleur de dialogue qu'elle avertit lorsqu'un événement se produit à son niveau. Par comparaison avec la théorie de la compilation, cette couche peut-être vue comme l'analyseur lexical d'un langage de programmation. C'est elle qui est chargée de dire si un élément donné appartient bien à l'alphabet du langage.

L'**Interface avec l'Application** est une surcouche de l'application. Elle correspond à tout ce que le contrôleur de dialogue connaît de l'application. C'est également elle qui est chargée de transformer les données de la couche de présentation, transmises par le contrôleur de dialogue, en données de l'application. Pour poursuivre la comparaison avec la théorie de la compilation, on peut voir cette couche comme la représentation de la sémantique d'un programme informatique.

Le **Contrôleur de Dialogue** fait le lien entre la Présentation et l'Interface de l'application. Il est prévenu par le premier dès qu'un événement survient. Il peut alors demander à la présentation de lui fournir certaines données, d'en afficher d'autres, etc. Il est également capable de demander à l'Interface de l'application d'effectuer certains traitements. Cette couche peut être vue comme l'analyseur syntaxique d'un compilateur : il est capable de dire si une séquence d'interaction constitue bien un mot du langage reconnu par l'application.

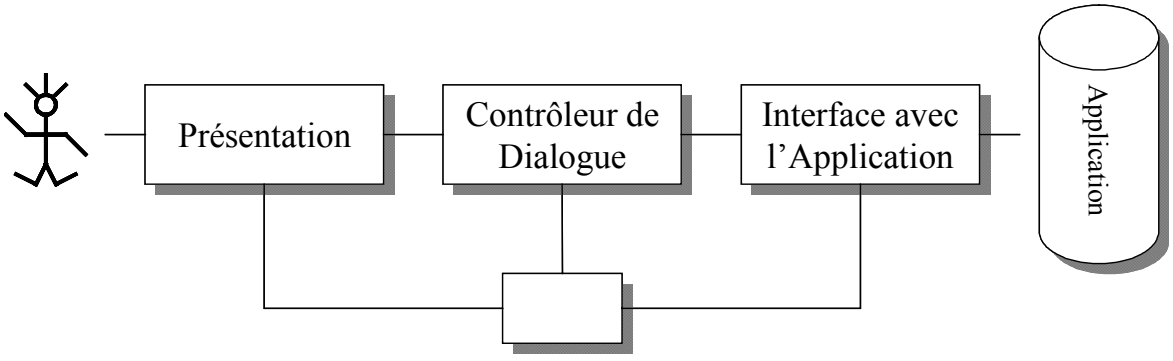


Figure 30: Le modèle SEEHEIM

Ce modèle a été le premier à décrire précisément la décomposition d'une application interactive en différents modules et à préconiser la séparation de l'interface homme machine proprement dite avec le reste de l'application. Il a été soumis à de nombreuses critiques,

notamment avec l'émergence de nouveaux concepts de programmation comme l'approche orientée objet, qui a mis en exergue son caractère monolithique. On lui a également reproché, comme aux autres modèles globaux, de ne pas décrire avec suffisamment de précision le fonctionnement du contrôleur de dialogue.

2.2.1.2 Le modèle ARCH

A la suite des travaux sur le modèle SEEHEIM, un certains nombres de séminaires réunissant des développeurs d'interface ont débouché sur le modèle ARCH [Bass, et al. 1991]. Dans ARCH, une application est divisée en cinq modules indépendants, tous décrits de manière beaucoup plus précise que dans SEEHEIM.

Le **Domaine** représente les objets et les méthodes propres à l'application elle-même. Ceux-ci sont implémentés de manière totalement indépendante des interactions avec l'utilisateur et ne tiennent pas compte de la façon dont les informations seront récupérées ou affichées. Dans SEEHEIM, par comparaison, rien n'interdisait formellement au concepteur de lier de manière directe l'interface avec l'application à la présentation.

L'**Adaptateur de Domaine** implémente les tâches relatives au domaine dans lesquelles l'utilisateur intervient. Il présente une surcouche des objets du domaine permettant ainsi leur manipulation au sein du contrôleur de dialogue.

Le **Contrôleur de Dialogue** a un rôle très proche de celui décrit dans SEEHEIM. Il autorise et contrôle l'enchaînement des interactions de la couche de présentation. C'est également lui qui déclenche les actions du Domaine via les commandes de la Présentation. Cependant, son rôle est détaillé de manière plus précise que dans SEEHEIM : il est ainsi en charge de maintenir la cohérence entre les différentes vues d'un même objet.

La **Présentation** est une surcouche logicielle du composant d'interaction. Alors que le **Composant d'Interaction** prend en compte les outils d'interactions, comme les Boîtes à Outils, la présentation permet de rendre le reste de l'application indépendante de toute plate-forme par une abstraction logique des objets et des méthodes d'interaction.

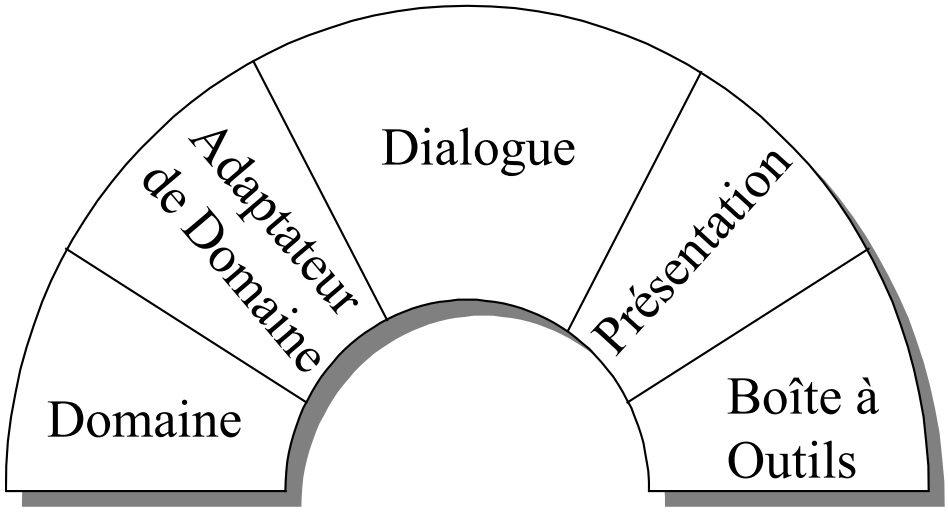


Figure 31 : Le modèle ARCH

Le modèle ARCH propose un découpage plus fin de l'application que ne le fait SEEHEIM. Cette décomposition permet à une application basée sur ce modèle de mieux supporter des modifications futures. Cependant, une des principales critiques à l'égard de cette architecture reste la description trop superficielle du fonctionnement des différentes couches, celui-ci étant sujet à de nombreuses interprétations.

2.2.1.3 Conclusion sur les modèles centralisés

Les modèles globaux (ou centralisés) ont été les premiers à proposer un découpage systématique d'une application interactive en différents modules, séparant notamment de manière explicite la partie « Applicative » de la partie « Interface ». Cette décomposition est fondamentale puisqu'elle définit une séparation des difficultés au travers de modules faisant l'objet de démarches méthodologiques spécifiques.

Le problème principal de ces méthodes est la description trop floue des différents composants identifiés. De ce fait, la modélisation d'un dialogue tel que celui qui nous intéresse, le dialogue structuré, doit être prise à part entière par le concepteur de l'application.

Les modèles centralisés ont vu leur développement lié à toute une série de travaux en matière de Génie Logiciel sur la modularité. L'essor des paradigmes de Programmation Orientée Objet (POO) ont fait tourner les efforts en architecture logicielle vers de nouveaux modèles. Là où la modularité préconisait une séparation nette des composants « applicatif » et « interface », les modèles répartis (ou multi-agent) utilisent les caractéristiques de la POO pour « éclater » l'application, chaque objet portant lui-même une part de la Présentation et du Noyau Fonctionnel.

2.2.2 Les modèles répartis ou multi-agents

L'avènement de la programmation orientée objet a incité les concepteurs d'architecture à utiliser la notion de stimuli-réponses. Les différents modèles répartis reposent donc sur le concept d'agent, dont le rôle et la composition varient d'une architecture à l'autre.

Un agent est ainsi capable de recevoir, d'émettre et de mémoriser des événements (ou des messages). Il peut enfin enregistrer son état et traiter des événements retenus.

Lorsqu'un agent émet un événement, celui-ci est envoyé, selon les modèles, ou bien à tous ses destinataires, ou bien à tous les agents dont les récepteurs sont actifs. Chaque agent peut alors traiter cet événement selon son propre « séquencement ».

2.2.2.1 Model - View - Controller (MVC)

MVC est le modèle d'architecture du langage Smalltalk [Goldberg 1984], repris plus récemment par Swing®, la boîte à outils du langage Java®. A l'intérieur de ce modèle, l'interface de l'application (voire l'application elle-même) est décomposée en un ensemble de trois modules indépendants capables de communiquer entre eux par message.

Le **Modèle** (Model) comprend les structures de données et les méthodes de l'application. Il conserve l'état de l'application ainsi que ses données. Il est le seul qui puisse communiquer avec d'autres modèles.

Le **Contrôleur** (Controller) gère donc les « entrées » de l'utilisateur transmises par l'intermédiaire de la vue et les communique au modèle.

La **Vue** (View) est la représentation externe du modèle (celle qui est affichée à l'écran). Le fait que le modèle et la vue soit dissociés permet à un modèle de représenter plusieurs vues.

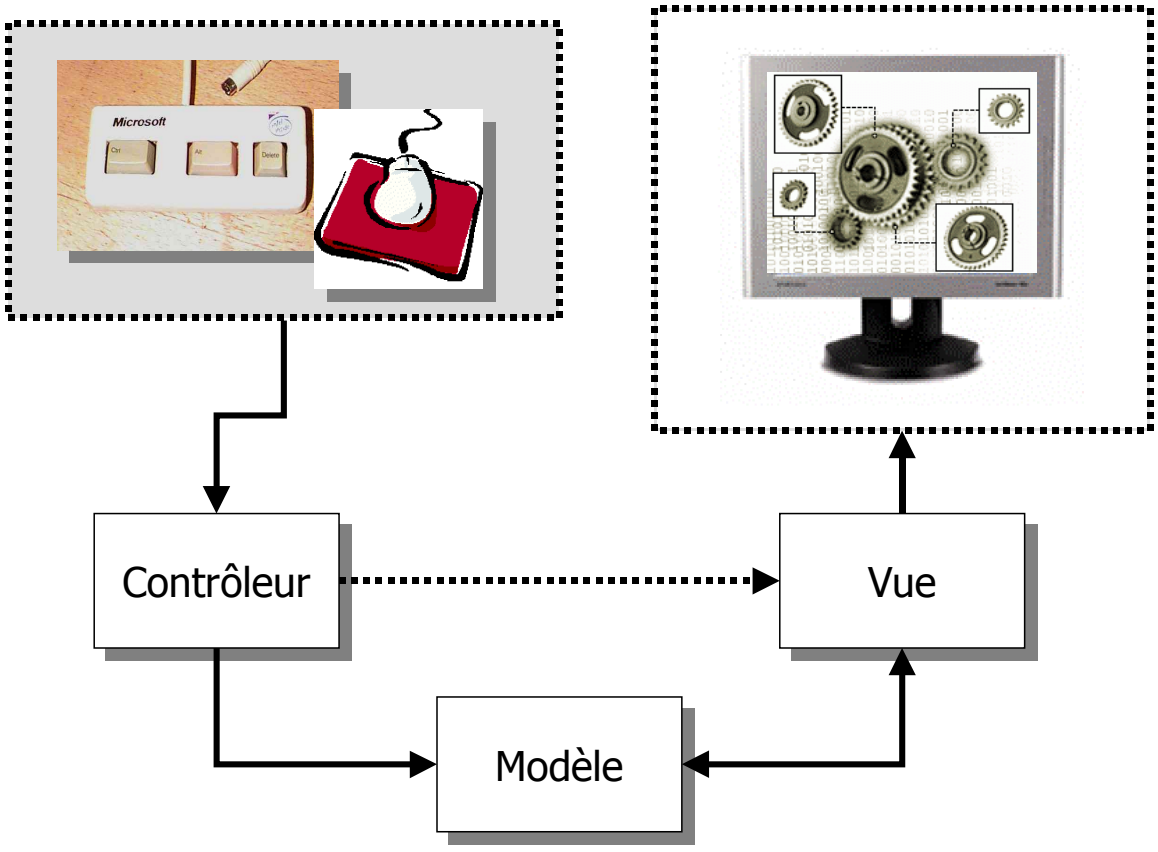


Figure 32 : Le modèle MVC

Le fonctionnement de cette architecture peut-être comparé à un modèle Client/Serveur, à la différence que les composants sont regroupés à l'intérieur d'une seule et même application. Lorsque l'utilisateur agit sur l'interface, le contrôleur envoie un message au modèle lui demandant d'accomplir l'action correspondante. Le modèle effectue l'opération et envoie à son tour un message à ses différentes vues, les informant d'une modification. Les vues peuvent alors interroger le modèle pour connaître ces modifications et mettre à jour la présentation en conséquence.

Le modèle MVC peut-être vu comme un modèle « SEEHEIM réparti » dans lequel la vue correspond à la présentation et le modèle correspond à l'interface avec l'application. Cette architecture offre l'intérêt de dissocier en trois instances indépendantes les trois composants de chaque objet d'interaction, et de séparer ainsi clairement la partie sémantique de la partie présentation. Ceci étant, cette architecture présente aussi quelques désavantages. Tout d'abord, le fait que les entrées passent systématiquement par le contrôleur peut nuire au caractère immédiat généralement souhaitable pour des échos. D'autre part le fonctionnement induit un échange très important de messages entre les différents composants, même pour des actions très simples. Enfin, on peut lui reprocher, comme aux autres modèles répartis, de ne pas définir un schéma méthodologique permettant d'analyser un problème et d'en déduire le nombre et la nature des agents à utiliser.

2.2.2.2 Présentation - Abstraction - Contrôle (PAC)

PAC a été mis au point en 1987 par Joëlle Coutaz [Coutaz 1987]. Tout comme MVC, on peut voir PAC comme une architecture SEEHEIM répartie, à la différence que PAC modélise l'application dans sa totalité, là ou MVC ne s'occupe « que » de l'architecture de l'interface. D'autre part, PAC est indépendant du langage d'implémentation.

Joëlle Coutaz dans [Coutaz 1990] décrit en ces termes les buts poursuivis par ce modèle :

- Définir précisément les composants d'un système interactif et décrire leur fonctionnement et leurs liens (modularité) ;
- Proposer un découpage en différents niveaux lexical, syntaxique et sémantique (modèle langage) ;
- Assurer des interactions simultanées avec l'utilisateur, à la fois en entrée et en sortie (multi-file d'activité).

PAC définit l'application de manière récursive, en proposant une hiérarchie d'agents, nommés agents PAC, composés chacun de trois composants :

L'**Abstraction** désigne l'ensemble des concepts et des fonctions de l'agent. Contrairement au modèle SEEHEIM, il ne s'agit pas d'une interface vers l'application mais d'une partie de l'application elle-même. L'Abstraction est la partie abstraite du concept manipulé et se trouve indépendante de toute représentation. Cela peut-être par exemple un objet dans une base de données, etc.

La **Présentation** définit l'image du système, c'est à dire son comportement en entrée comme en sortie vis-à-vis de l'utilisateur. A la différence de MVC, un composant unique

remplit ce double rôle, évitant que les messages transitent par un composant intermédiaire et ralentissent, par exemple, les échos. La Présentation visualise l'état interne du composant Abstraction, c'est-à-dire l'état du système. C'est ce constituant qui, via la boîte à outils, est en contact avec l'utilisateur pour afficher les informations de l'Abstraction et lui communiquer, via le Contrôleur, les actions de l'utilisateur.

Le **Contrôle** entretient la cohérence entre la Présentation et l'Abstraction et peut même jouer le rôle de traducteur pour permettre le changement de formalisme de l'un vers l'autre (celui de l'Abstraction n'étant pas toujours adéquat pour l'utilisateur et donc la Présentation). Il est également en charge de la résolution des conflits, des synchronisations et des rafraîchissements.

L'exemple classique considère un objet interactif de saisie d'angle sur lequel l'utilisateur peut interagir. L'abstraction de cet objet comprend les données relatives au concept manipulé : angle maximal, angle minimal et valeur courante. La présentation affiche un objet d'interaction qui peut être manipulé par l'utilisateur. Si celui-ci modifie l'angle, un message est envoyé au Contrôleur qui traduit la valeur en une valeur compréhensible par l'Abstraction et la lui transmet.

En dehors des avantages flagrants présentés par les agents « élémentaires » (par exemple, l'indépendance de la Présentation et de l'Abstraction entre elles leur confère un statut hautement réutilisable), l'architecture PAC présente un fonctionnement hiérarchique qui en fait un modèle à part.

Les agents PAC peuvent en effet être organisés en une hiérarchie arborescente (Figure 33), un agent pouvant être composé de plusieurs agents fils le spécialisant. Un agent parent a la responsabilité des agents fils, par conséquent le niveau d'abstraction de chaque agent est proportionnel à son niveau dans la hiérarchie. Un agent peut ainsi en utiliser plusieurs autres : lorsque sa présentation est modifiée, un contrôle prévient son précédent dans la hiérarchie, lequel, à son tour, avertit les autres subordonnés, afin qu'ils reflètent le changement dans leur propre présentation. Cette relation hiérarchique est une relation de composition et non une relation d'héritage au sens objet du terme (pas d'héritage de propriété). Il faut noter également que la communication entre agents passe par le Contrôle qui se trouve par conséquent au centre du dialogue de l'agent.

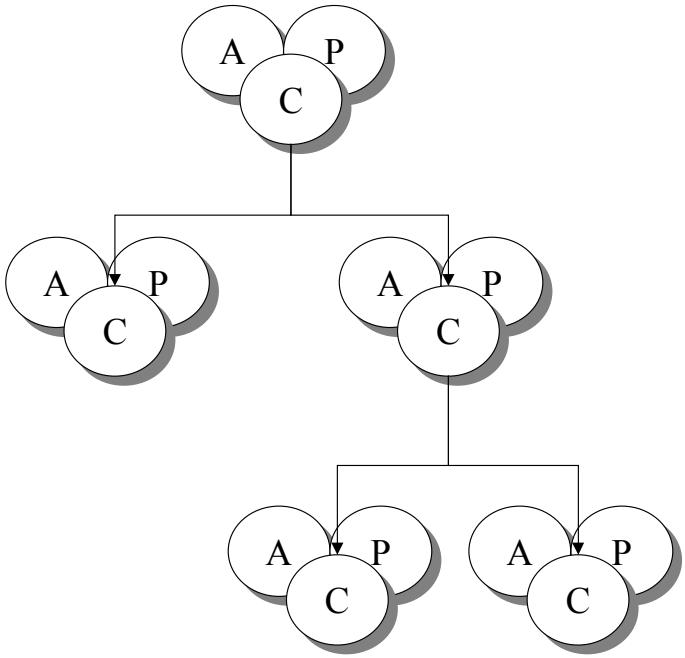


Figure 33 : Hiérarchie d'agents PAC

Dans le modèle PAC, l'application elle-même est un agent élémentaire. Ce modèle constitue ainsi un support de conception d'une interface utilisateur à tous les niveaux d'abstraction.

Même si « la décomposition peut se faire avec une granularité aussi petite que l'on veut » ([Fekete 1996]), le modèle PAC amène de manière naturelle le concepteur à associer tout objet du domaine à un objet interactif visualisable et manipulable par l'utilisateur. Ceci étant, cette approche n'est envisageable que si les objets du domaine sont autonomes ou relativement statiques. Dans le cas contraire, aucune piste n'est donnée quant à l'identification des agents ([Coutaz 1992]).

Le point fort de cette architecture logicielle est sa portabilité. De la séparation nette entre le Noyau Fonctionnel et la partie Présentation résulte un isolement des composants logiciels proches de l'implémentation de l'interface (boîte à outils). La modification de l'interface implique donc une restructuration complète des composants « Présentation » des agents ainsi que des modifications profondes du « Contrôleur », responsable de l'interface entre Présentation et Abstraction. Par contre, l'Abstraction se trouve, elle, complètement épargnée par ces altérations.

On peut reprocher au modèle PAC [Patry 1998] les conséquences néfastes de sa structure hiérarchique sur la transmission des messages. Lorsque le nombre de niveaux devient important, il peut être coûteux en temps de parcourir la structure de couche en couche. D'autre part, comme nous l'avons déjà évoqué, cette architecture ne propose pas une identification claire en nombre et en rôle des différents agents par rapport à un problème donné.

2.2.2.3 Conclusion sur les modèles répartis

Les modèles multi-agents représentent les différents composants déjà identifiés dans les modèles globaux (abstraction, présentation, contrôle) mais à un niveau de granularité beaucoup plus faible. Ils ont également en commun de modéliser tous les aspects d'un objet (domaine, visualisation et contrôle) par une instance spécifique, un objet abstrait étant représenté par un agrégat de plusieurs instances. Au contraire des méthodes globales, l'application est « éclatée », chaque objet portant une part de la Présentation et du Noyau Fonctionnel. Ces modèles présentent l'avantage de reposer sur les paradigmes de programmation orientée objet et donc de correspondre à un mode de conception « moderne ».

Le reproche principal que l'on peut adresser à ce type d'architecture est de ne pas établir clairement les domaines d'application auxquels il s'adresse et de ne pas identifier de manière précise le rôle et le nombre des agents qui doivent composer une application interactive [Coutaz 1990].

D'autre part, si cette architecture se prête très bien aux applications manipulant des objets très peu relationnels et des tâches atomiques mono-objet, il n'en est pas de même pour les applications mettant en jeu des objets fortement liés et des tâches structurées. En effet, dans ce dernier cas, la structure des tâches correspond rarement à celles des objets de l'application et le concepteur se retrouve livré à lui-même car aucun support n'existe pour le guider dans la modélisation de son dialogue.

Les modèles répartis ne constituent donc pas, de ce fait, une solution viable pour les applications que nous voulons mettre en place et qui, rappelons-le, rentrent dans la catégorie des systèmes multi-objet, dont les tâches sont structurées et dont les objets sont composés et relationnels.

La dernière section de cette partie s'intéresse aux modèles hybrides, qui concilient la décomposition modulaire des modèles globaux et une description fine du fonctionnement des différents composants.

2.2.3 Les modèles hybrides

Contrairement aux modèles dits « globaux », les modèles multi-agents définissent à un niveau de granularité très faible le fonctionnement des différents composants qui doivent composer une application. Malheureusement, cette description s'est faite aux dépens de la distinction des composants identifiés à un niveau macroscopique par les modèles globaux.

L'approche « hybride » tente de réconcilier ces deux techniques en reprenant les avantages de chacune. Nous présentons ici PAC Amodeus et le modèle Multi-couches avant de nous attarder plus longuement sur l'architecture que nous avons choisi d'utiliser, H[4].

2.2.3.1 PAC - Amodeus

Les modèles multi-agents proposent de structurer toute une application interactive (que ce soit son noyau ou sa présentation) à partir de triplets d'instance. Cette méthode impose au concepteur de développer en même temps l'application et son interface avec l'utilisateur. Malheureusement, cette technique de conception ne peut convenir que pour certaines classes d'applications. D'autre part, elle ne permet en aucune façon de rendre interactif un noyau fonctionnel préexistant.

Pour ces raisons, Joëlle Coutaz et Laurence Nigay dans un premier temps avec PAC-SEEHEIM [Coutaz & Nigay 1991], puis avec PAC AMODEUS [Nigay & Coutaz 1991], ont cherché à se baser sur l'architecture ARCH pour bénéficier de l'indépendance entre le Domaine et la Présentation que propose celle-ci.

Dans PAC AMODEUS, l'application reprend ainsi le découpage en cinq composants décrit dans ARCH. La différence fondamentale vient du fait que le Contrôleur de Dialogue est lui-même décomposé en une hiérarchie d'agents PAC (Figure 34). Ces agents sont chargés de la communication avec l'adaptateur de Noyau Fonctionnel par des objets représentant une abstraction des objets du domaine (traduction de formalisme entre les couches du domaine et celle de la présentation, association entre les données de l'adaptateur de Noyau Fonctionnel et l'interface de l'application). Ils s'occupent également du séquencement des tâches.

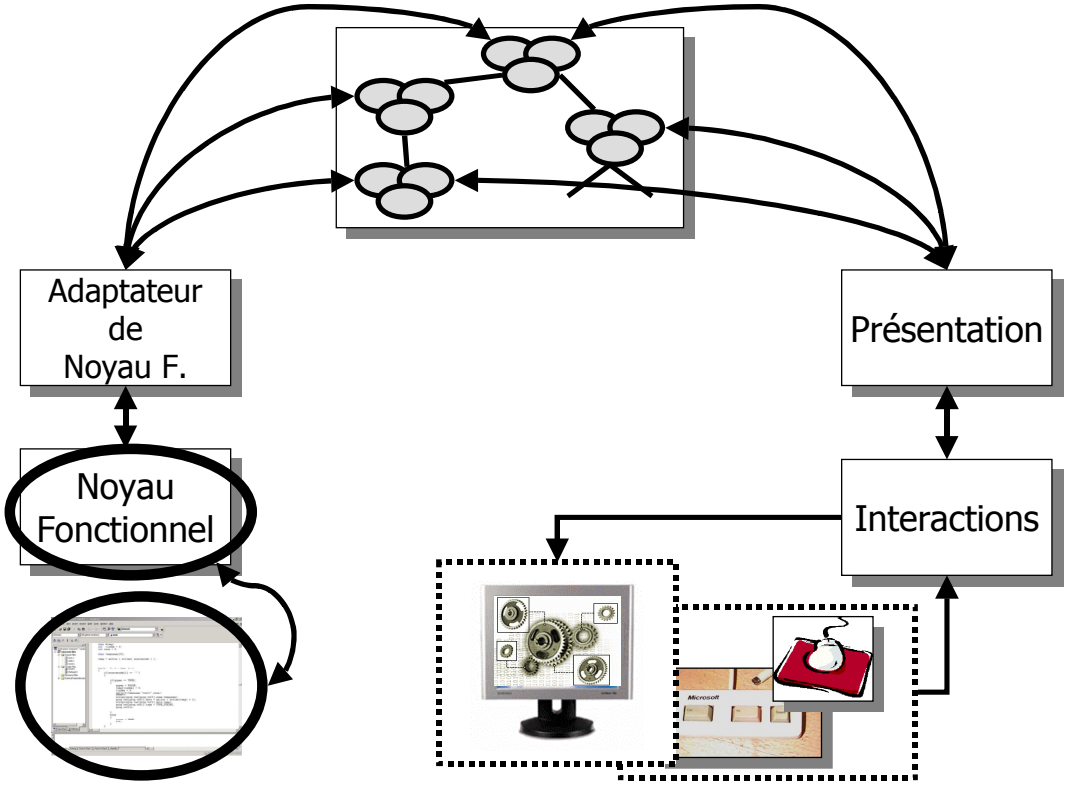


Figure 34 : PAC AMODEUS

Cette structuration du Contrôleur de Dialogue est totalement indépendante du Noyau Fonctionnel et elle permet par conséquent de rendre interactif un Noyau Fonctionnel pré-

existant (comme dans MATIS [Nigay 1994] par exemple). D'autre part les techniques de Fusion/Fission, telles qu'elles sont décrites dans [Nigay & Coutaz 1993], permettent de prendre en compte des dialogues multi-modaux.

Malgré cela, PAC AMODEUS paraît conserver la grande imprécision déjà analysée dans [Coutaz 1990] pour organiser les agents du contrôleur de dialogue. Certains agents modélisent des objets très proches de la présentation (fenêtre de travail), d'autres établissent des correspondances bijectives entre les objets du domaine et ceux de la présentation, d'autres enfin sont destinés à représenter des tâches du système. Cette multiplicité des rôles rend difficile leur organisation. Pour cette raison, un système expert a été mis au point afin d'automatiser la génération du Contrôleur de Dialogue [Nigay 1994].

D'autre part, ce modèle prévoit un flot de données symétrique : les données transitent de l'utilisateur vers le Noyau Fonctionnel en passant par le Contrôleur de Dialogue, et inversement. Or, cette symétrie ne paraît pas toujours justifiée car, dans une application interactive, l'utilisateur et le Noyau Fonctionnel ne fournissent pas des données de même nature. Comme c'est le cas dans le modèle de SEEHEIM, il semble plus cohérent que l'adaptateur de domaine puisse accéder de manière directe à la couche de présentation pour rendre compte de son état sans passer par le Contrôleur de Dialogue.

Du point de vue de la mise en place d'un Dialogue Structuré, le problème principal vient du fait que PAC AMODEUS ne permet pas de séparer les tâches sémantiques, propres au Domaine, des tâches articulatoires, spécifiques à la Présentation.

2.2.3.2 Modèle multi-couche

Dans sa thèse [Fekete 1996], Jean-Daniel Fekete présente un modèle d'architecture original, spécialisé pour les applications interactives de type MacDraw. L'auteur part du principe que les différents objets graphiques sont de nature très différentes et que, si certains sont inertes (fond de l'écran), d'autres changent parfois (objet sélectionné quand il est manipulé avec la souris) ou souvent (vidéo affichée). Ainsi, plutôt que de gérer les objets graphiques au sein d'un même niveau, il propose d'utiliser autant de niveaux que nécessaire pour séparer les éléments graphiques qui suivent un comportement spécifique.

L'interface est donc découpée en différentes couches qui peuvent gérer les événements, gérer les objets graphiques ou améliorer les performances. Ces couches sont projetées sur une surface virtuelle pour composer l'interface graphique présentée à l'utilisateur. La Figure 35, inspirée de [Fekete 1996], montre l'ordre dans lequel les couches sont composées dans une pile pour obtenir une image finale. Les couches s'affichent du fond vers le premier plan tandis que les événements suivent le sens inverse. Sur l'exemple de la Figure 35, une couche est chargée d'afficher une grille, une autre s'occupe de la sélection, la suivante de l'affichage des objets graphiques et une dernière du fond de l'écran.

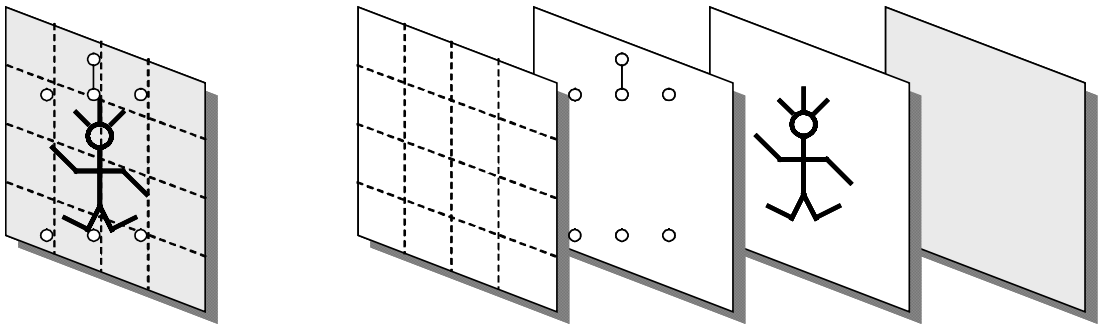


Figure 35 : Les différentes couches possibles d'une application basée sur le modèle de Jean-Daniel Fekete

Dans cette architecture, les couches communiquent par événement. Lorsqu'un événement utilisateur est produit, il parcourt successivement les couches de l'interface jusqu'à ce que l'une d'entre elle accepte de le traiter. C'est alors un outil spécifique de la couche qui est chargé de ce traitement.

Il faut noter que si les couches ne se connaissent pas entre elles, les outils de différentes couches peuvent faire appels les uns aux autres pour élaborer des tâches complexes. Ils sont ainsi capables, par exemple, d'initier une phase de dialogue gérée par une autre couche. Lorsque la couche gérant le fond de l'écran reçoit un click souris, elle peut en déduire qu'aucune autre couche ne l'a intercepté et que le dialogue se trouve dans un état tel que ce click représente le début d'un rectangle de sélection. Elle peut donc initier cette tâche même si la sélection proprement dite sera gérée par une autre couche.

La décomposition en couches est intéressante pour les applications graphiques, car elle permet de répartir le dialogue sur plusieurs niveaux. D'autre part, cette architecture permet un raffinement des événements : une couche peut ainsi transformer les coordonnées d'une position de curseur pour qu'elles restent dans un repère donné, ou encore transformer un événement en un autre type d'événement qui sera transmis aux couches suivantes.

Cependant, si les couches sont par nature indépendantes les unes des autres, et qu'elles n'ont besoin de connaître ni l'origine ni la destination des événements qu'elles reçoivent et émettent, il n'en n'est pas de même pour les outils. Ceux-ci sont fortement liés les uns aux autres puisque, pour activer la tâche d'une autre couche, un outil doit connaître l'existence de l'outil correspondant. Comme l'analyse Guillaume Patry dans [Patry 1999], la gestion de ces outils, quand leur nombre et leurs interactions augmentent, peut vite s'avérer un frein au développement d'une application utilisant cette architecture.

2.2.3.3 Conclusion

Les modèles hybrides que nous venons de voir constituent deux solutions très intéressantes en répondant aux attentes laissées en suspend par les modèles globaux et répartis. En utilisant une décomposition globale et en décrivant de manière plus précise le rôle et le fonctionnement des composants, elles offrent un support considérable à la conception d'une application interactive.

Malgré ça, l'une comme l'autre restent inadaptées à la modélisation d'un dialogue structuré. PAC Amodeus, nous l'avons vu, reste trop imprécis sur l'identification des agents du Contrôleur de Dialogue et ne permet pas la décomposition des tâches sémantiques et articulatoires. Le modèle Multi-couches, quant à lui, présente l'inconvénient de lier de manière trop forte les différents outils de couches de son modèle. Dans le cas de tâches complexes, ces interactions fortes peuvent devenir difficilement gérables.

2.2.4 Conclusion partielle

Le Dialogue Structuré est un mode de dialogue très utilisé dans les Applications Graphiques Interactives de Conception Technique, et notamment dans les systèmes de CAO. Il permet à un utilisateur d'exprimer des tâches complexes de manière précise, en décomposant celle-ci en un ensemble de sous-tâches, plus faciles à atteindre.

En dépit de l'importance que revêt ce concept, aucune architecture logicielle existante ne permet de prendre en compte explicitement ce type de dialogue, et aucune n'offre un support suffisant pour permettre sa mise en place de manière aisée.

La section suivante présente l'architecture H4, une architecture hybride reposant sur le modèle ARCH et dont le Contrôleur de Dialogue possède un fonctionnement propre à la modélisation de dialogues structurés.

2.2.5 H4

H4 est une architecture logicielle qui a été développée au laboratoire du LISI par Patrick Girard, Laurent Guittet et Guy Pierra ([Guittet & Pierra 1993c], [Girard, Pierra, & Guittet 1995], [Guittet 1995]). Sa conception repose sur le besoin des AGICT, et notamment les systèmes de CAO, de disposer d'un dialogue structuré. Ce modèle est basé sur ARCH et distingue également cinq composants logiciels. Mais, à la différence de ce dernier, H4 décrit de manière précise tous les modules et préconise la structuration de quatre d'entre eux en une hiérarchie d'agents.

2.2.5.1 Macro composants

Le modèle H4 est composé, de la même manière que ARCH, de cinq composants (Figure 36) :

- le **Noyau Fonctionnel** représente les objets du domaine ainsi que toutes les fonctionnalités qui y sont associés. Il est la véritable sémantique de l'application ;
- l'**Adaptateur de Noyau Fonctionnel** est une surcouche logique du Noyau Fonctionnel. Il s'agit d'une interface entre le Contrôleur de Dialogue et le Noyau Fonctionnel qui ces deux éléments indépendants entre eux. Il est donc chargé de traduire les informations du Noyau Fonctionnel en données compréhensible par le Contrôleur de Dialogue et inversement. D'autre part, il est capable de communi-

quer avec l'adaptateur de présentation pour afficher l'état du Noyau Fonctionnel, quand le passage par le Contrôleur de Dialogue n'a pas de réelle signification ;

- L'**Adaptateur de Présentation** est chargé de rendre l'application indépendante de l'implémentation de l'interface graphique en mettant à la disposition du Contrôleur de Dialogue une surcouche des objets et des primitives de la Présentation (en fait, la Boîte à Outils). Il fournit donc à l'application les outils permettant de réaliser l'affichage de l'état du Noyau Fonctionnel ;
- La **Présentation** gère les entrées et les sorties avec l'utilisateur. Par l'intermédiaire de composants d'interaction (les widgets d'une Boîte à Outils), cette couche logicielle récupère les informations fournies par l'utilisateur et affiche l'état du Noyau Fonctionnel. Ce composant correspond à la Boîte à Outil de ARCH.

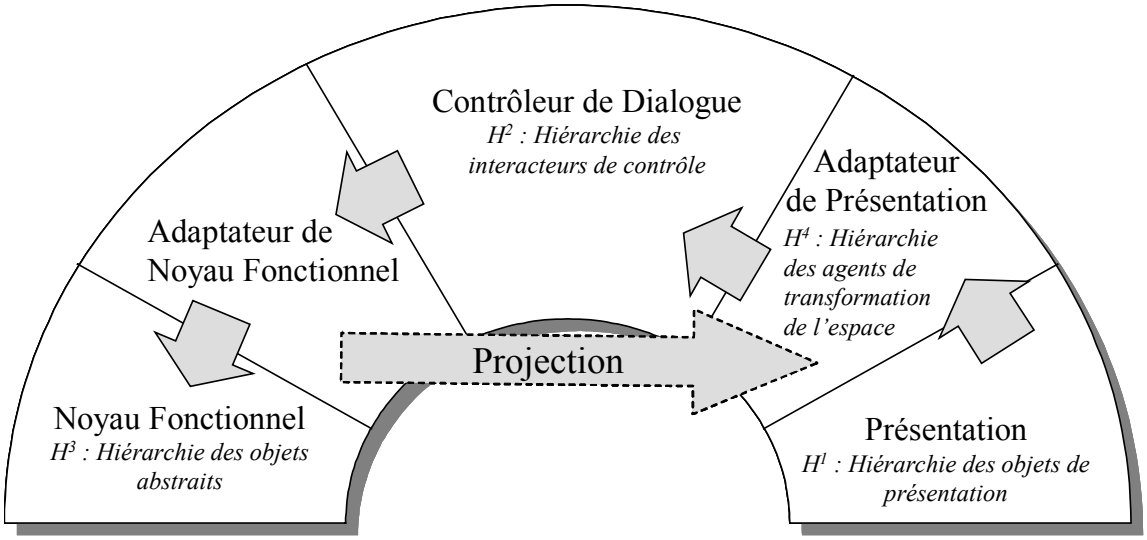


Figure 36 : les différentes couches du modèles H^4

2.2.5.2 Dialogue structuré

Comme nous l'avons vu dans la première partie de ce chapitre, Norman [Norman 1986] préconise la décomposition d'un but de haut niveau en une hiérarchie de sous-buts plus faciles à atteindre pour l'utilisateur. Cette division correspond à la notion de tâche structurée, dans laquelle une tâche peut s'exécuter dans le contexte d'une autre tâche afin de lui fournir un résultat, utilisable comme paramètre.

Un tel mécanisme n'est envisageable que dans le cadre d'un dialogue préfixé, dans lequel l'utilisateur indique d'abord le nom de la commande à effectuer AVANT de fournir les paramètres d'exécution. L'expression ainsi constituée représente le Dialogue Structuré.

De cette logique découle naturellement la notion de production/consommation et une hiérarchisation des tâches. Certaines tâches produisent des informations qui sont récupérées et utilisées comme paramètres par d'autres. Cela induit que le contrôleur de dialogue de l'application doit être capable de différencier les tâches de production (qui fournissent des informations) des tâches de consommation (qui utilisent des informations produites

par d'autres tâches). Ainsi, nous appelons **tâches terminales** les tâches identifiées comme pouvant être associées à des objectifs de haut niveau (au sens de Norman) de l'utilisateur. Celles-ci modifient l'état du Noyau Fonctionnel mais ne produisent pas de données. D'autre part, nous appelons **sous-tâches de production** les tâches calculant et fournissant un résultat à d'autres tâches. Les sous-tâches de production se placent, dans la hiérarchie, à un niveau inférieur à celui des tâches terminales.

Les différentes tâches d'un système doivent être regroupées en différents niveaux d'abstraction, suivant les services qu'elles fournissent à l'application. Dans la plupart des systèmes, on distingue au moins trois niveaux :

- Les tâches de création, qui modifient l'état du système en créant de nouveaux objets ;
- Les tâches de calcul, qui sont chargées de récupérer la valeur de l'attribut d'un objet ou d'effectuer un certain nombre de calculs sur ces attributs ;
- Les tâches de sélection, qui permettent de fournir une entité du modèle (sélectionnée par exemple avec le curseur de la souris) aux autres tâches.

Le dernier élément primordial dans la constitution d'un dialogue structuré concerne l'information qui transite de tâche productrice en tâche consommatrice. Celle-ci est identifiée par son type et est indépendante de son mode de production. Comme l'analyse Guillaume Texier [Texier 2000], l'expression d'un dialogue structuré nécessite donc :

- L'identification des unités d'information indépendamment de leur mécanisme de production. De telles unités d'information sont appelées **jetons** ;
- La description des tâches en termes de production/consommation et leur affectation à un niveau d'abstraction, ce qui définit la syntaxe du langage de dialogue reconnu par le contrôleur de dialogue.

Nous avons récapitulé ici les différents éléments qui constituent un dialogue structuré. Pour qu'une architecture supporte ce type de dialogue, il faut que son Contrôleur de Dialogue :

- Soit doté d'une unité d'information qui transite d'une tâche à l'autre ;
- Identifie les tâches de production et les tâches terminales ;
- Permette le regroupement des tâches en différents niveaux d'abstraction ;
- Organise les différents niveaux d'abstraction de manière hiérarchique.

Le Contrôleur de Dialogue de H4 présente toutes ces caractéristiques. La partie suivante détaille son fonctionnement.

2.2.5.3 Contrôleur de dialogue

Dans ce paragraphe, nous détaillons les différents éléments qui rentrent en jeu dans le contrôleur de dialogue de H4 puis nous détaillons son fonctionnement.

2.2.5.3.1 Jeton

L'unité d'information est le **jeton**. On distingue les **jetons paramètres** des **jetons commandes**. Les premiers représentent une abstraction des données du Noyau Fonctionnel et de la Présentation. Un jeton paramètre transporte ainsi des valeurs (une position graphique donnée par l'utilisateur sera transportée par un jeton « position » stockant une abscisse et une ordonnée). Les jetons commandes, quant à eux, représentent l'intention de l'utilisateur. Ils transportent l'identifiant d'une tâche à activer.

2.2.5.3.2 Tâches

Dans H4, les tâches sont représentées par des signatures de fonction appelées **Questionnaires**. Un questionnaire est défini par son nom, ses paramètres d'entrée et, éventuellement, un paramètre de sortie. Elles sont implémentées dans l'Adaptateur de Noyau Fonctionnel.

Ces modules constituent le lien entre le Contrôleur de Dialogue et les primitives de l'application (par le biais de l'adaptateur de Noyau Fonctionnel).

2.2.5.3.3 Interacteurs

Comme nous l'avons vu précédemment, les tâches remplissant des services similaires peuvent être regroupées à l'intérieur de niveaux d'abstraction communs. Un interacteur, comme il est défini dans [Guittet & Pierra 1993a], [Guittet & Pierra 1993b] et [Guittet 1995], représente un niveau d'abstraction. Il réunit donc des questionnaires (correspondant à des tâches).

Les interacteurs de H4 ne sont pas les mêmes que les interacteurs d'entrée / sortie décrits dans [Harrison & Duce 1994] ou [Paternò & Faconti 1994], ou que ceux que Brad Myers utilise [Myers, et al. 1990] dans son système GARNET. Ils gèrent les appels des primitives du système, en fonction des entrées de l'utilisateur, tandis que les interacteur d'entrée / sortie sont chargés des échanges entre le système et l'utilisateur.

Les différentes implémentations de H4 réalisées jusqu'à présent prennent comme modèle un réseau de transitions augmenté (ATN pour Augmented Transition Network [Woods 1970]). Un ATN peut-être vu comme un automate à états fini comprenant une variable d'état, appelée registre. Dans cet automate, les transitions s'effectuent sur des jetons : quand l'interacteur reçoit un jeton d'un type donné, il « regarde » si son état courant propose une transition vers un autre état par ce type de jeton. Si c'est le cas, il franchit la transition et stocke la valeur du jeton accepté dans son registre. Lorsqu'il reçoit un jeton qui le renvoie dans son état initial, il appelle le questionnaire correspondant, lui transmet toutes les valeurs stockées et vide son registre (Figure 37).

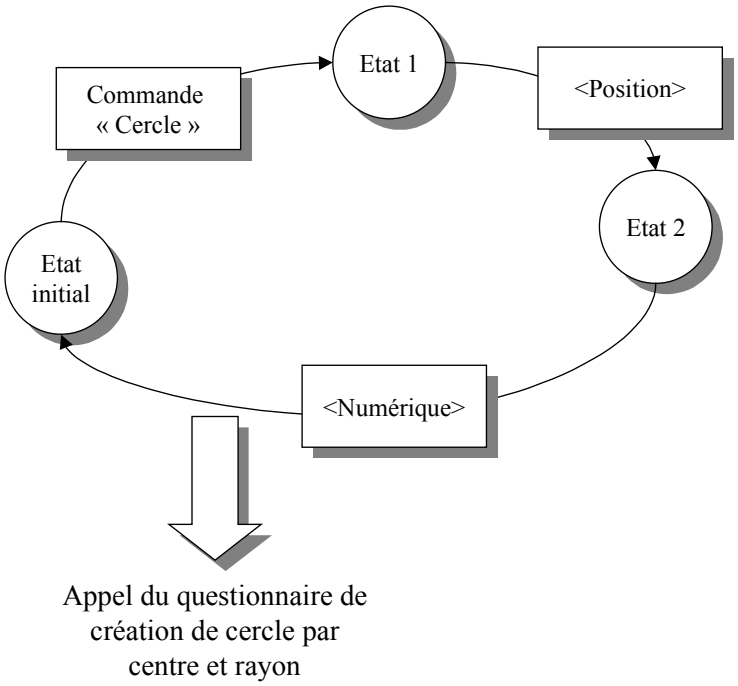


Figure 37 : Automate représentant un questionnaire

Dans le cadre d'une application graphique interactive de conception technique utilisant un dialogue préfixé, l'état initial de l'interacteur ne comporte qu'une seule transition, franchissable uniquement par un jeton commande. Cela correspond à contraindre l'utilisateur à fournir d'abord le nom de l'opération à effectuer avant d'en donner les opérandes.

2.2.5.3.4 Moniteur

L'organisation hiérarchique des interacteurs (donc des questionnaires et plus généralement des tâches) est à la charge d'un composant logiciel : le **moniteur**. Ceux-ci sont classés de bas en haut en suivant leur niveau d'abstraction (les interacteurs de plus bas niveau étant placés sous ceux de niveau plus élevé).

Le moniteur récupère les jetons de la couche de présentation et les transmet successivement à tous les interacteurs, en commençant par celui de plus bas niveau.

2.2.5.3.5 Fonctionnement

Lorsque l'utilisateur effectue une action, l'adaptateur de présentation transforme la donnée reçue de la présentation en un jeton d'un type correspondant contenant une valeur (par exemple, un click de souris peut être transformé en un jeton de type « position » contenant les coordonnées du point de l'écran sur lequel l'utilisateur à cliqué). Celui-ci est transmis au Moniteur qui le présente au premier interacteur de sa hiérarchie. L'interacteur peut alors l'accepter, si l'état courant de l'automate qui le représente possède une transition sur ce type de jeton. Dans ce cas, l'automate change d'état et stocke la valeur du jeton. D'autre part, si la transition replace l'automate dans son état initial,

l'interacteur appelle le questionnaire correspondant en lui transmettant toutes les valeurs enregistrées. Celui-ci est ensuite chargé de faire le lien avec le Noyau Fonctionnel en transformant les jetons en données du domaine et en appelant la primitive associée. Dans le cas où cette primitive produit un résultat (cas d'une tâche productive), la valeur calculée est à son tour transformée en jeton puis rendue au moniteur qui se charge de le transmettre aux interacteurs suivants dans la hiérarchie (Figure 38).

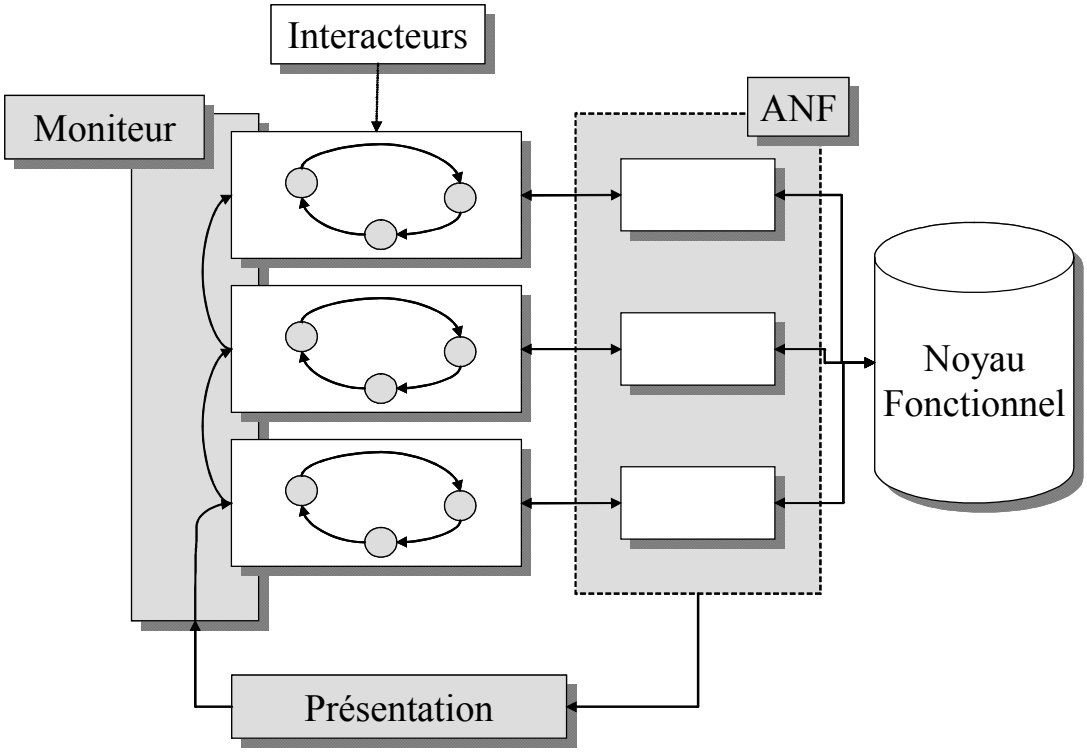


Figure 38 : Fonctionnement du Contrôleur de Dialogue de H[4]

Dans le cas où l'automate de l'interacteur ne se trouve pas dans un état propre à activer une transition par le jeton proposé, il le rend au moniteur qui le propage à l'interacteur suivant dans la hiérarchie, et ainsi de suite jusqu'à ce qu'il soit consommé ou qu'il n'y ait plus d'interacteur.

Cette organisation et ce fonctionnement assurent l'indépendance des tâches. Aucun interacteur ne connaît ni la source, ni la destination des jetons qu'il reçoit et qu'il émet. Ceci assure une parfaite modularité et permet à tout interacteur d'intervenir dans la construction d'une phrase du dialogue, sans que cela ait été prévu par le concepteur de l'application.

L'étude du fonctionnement de H[4] montre bien l'importance de la hiérarchisation des tâches : il faut être vigilant dans le placement des interacteurs à l'intérieur de la hiérarchie. Si I_1 et I_2 sont deux interacteurs, rangés dans cet ordre dans la hiérarchie, acceptant tout deux une entrée de type T, un jeton de type T accepté par I_1 ne sera jamais proposé à I_2. Considérons par exemple une application mettant en scène une calculette graphonumérique. On veut calculer le résultat de la multiplication d'un nombre n par la distance séparant deux pointés graphiques. L'ambiguïté va porter sur les paramètres d'entrée des

interacteurs chargés du Zoom (deux coordonnées spatiales) et du calcul d'une distance (également deux coordonnées spatiales). Si on exécute la série d'instructions suivante :

<Calcul> n ✲ <Distance> ➚ <Zoom> ➚ ➚ ➚

<> représente une commande

n est un entier

✲ représente la multiplication

➚ représente un pointé graphique

La question est de savoir comment la séquence va être interprétée : le système va-t-il considérer les deux pointés graphiques suivant la commande <Zoom> comme les deux entrées nécessaires à l'activation du zoom ou bien va-t-il utiliser le deuxième pointé pour activer la fonction de calcul de distance à laquelle on a déjà fourni un paramètre ? Ceci est entièrement déterminé par la hiérarchie dans le moniteur : si l'interacteur permettant le calcul de la distance est placé avant celui chargé du zoom, le calcul du produit aura lieu avant le zoom.

2.2.5.4 Conclusion

Contrairement à d'autres architectures hybrides, comme par exemple PAC AMODEUS, H[4] ne possède pas un caractère universel et ne permet pas la modélisation de toute forme de dialogue. En contrepartie, sa spécialisation lui donne la possibilité de décrire de manière précise un contrôleur de dialogue autorisant une modélisation simple d'un dialogue structuré. Cette particularité en fait une architecture privilégiée pour les AGICT.

2.3 Conclusion

Au cours de cette partie, nous avons étudié un certain nombre de modèles d'architecture. Qu'ils soient globaux, répartis ou hybrides, tous tentent de faire en sorte que la conception d'une application graphique soit moins coûteuse en temps, en effort et en maintenance. Pour cela, ils préconisent tous de séparer le Noyau Fonctionnel, la Présentation et le Contrôleur de Dialogue.

Alors que les modèles dits « globaux » s'appuient sur une description macroscopique de l'application, en décomposant celle-ci en différents composants, les modèles répartis s'inspirent de l'approche objet pour « éclater » les éléments de Présentation et de Contrôle de Dialogue à travers toute l'application. Si les premiers ne décrivent pas suffisamment le fonctionnement des différents éléments, les seconds manquent d'un cadre plus formel pour l'identification du nombre et du rôle des agents nécessaires selon l'application. L'approche des modèles hybrides tend elle à prendre en compte l'approche globale tout en essayant de décrire précisément le fonctionnement des différentes unités.

Dans la mesure où nous nous intéressons à des Applications Graphiques Interactives de Conception Technique, cadre dans lequel rentrent les applications de simulations de métamorphoses, nous avons étudié ces architectures sous l'angle du Dialogue Structuré. Or le résultat final est que, malgré l'importance du concept, dont l'utilisation s'avère essen-

tielle dans les AGICT, aucune approche existante n'évoque ni ne détaille une manière de le prendre en considération. Le concepteur d'un tel système doit donc assumer seul la lourde tâche de modélisation du dialogue.

L'architecture H[4] a été mise au point dans l'objectif de répondre à ce besoin en expliquant comment construire un Contrôleur de Dialogue capable de prendre en compte facilement le Dialogue Structuré. C'est donc tout naturellement que nous nous basons sur elle dans le cadre de nos travaux sur la mise en place d'un environnement de développement d'applications de métamorphoses.

Dans la partie suivante, nous poursuivons notre étude sur les outils de conception, et nous analysons les différentes techniques existantes pour créer de manière intuitive des programmes dont les caractéristiques sont semblables à celles des algorithmes décrivant les métamorphoses.

3 Programmation Interactive

3.1 Introduction

Dans les années soixante, l'avènement naissant des systèmes informatiques grand public permettait de penser que l'ordinateur allait se mettre à la portée de tous, et devenir à ce point universel qu'il serait bientôt un médium aussi incontournable que le livre. A ce tire, il était naturel de penser que, comme il est indispensable de savoir lire et écrire un livre, il serait bientôt nécessaire d'être capable de « lire » et « écrire » à l'intérieur d'un ordinateur. Selon Alan Kay [Cypher 1993], « lire » revient à comprendre des messages représentés par des signes dans un médium utilisant des conventions définies entre auteur et lecteur. La « Lecture » d'un ordinateur requiert donc un langage « d'interface utilisateur », compréhensible par tous les utilisateurs et capable, pour les concepteurs, de couvrir de manière universelle l'expression de tous les domaines possibles. Au contraire, « Ecrire » dans un ordinateur, c'est-à-dire le programmer, est une tâche beaucoup plus complexe qui exige d'un **utilisateur final** la capacité d'exprimer le même « genre de choses » que celles qu'il a l'habitude de lire.

Pour créer un programme, il faut utiliser un langage de programmation et assembler de manière textuelle les différents éléments mis à la disposition du concepteur, tout en respectant une syntaxe souvent rébarbative. Il faut également être capable de manipuler des concepts aussi abstraits que la variable, la primitive, les structures de contrôle, etc. Enfin, le programmeur doit réaliser de complexes gymnastiques de l'esprit pour réussir à projeter mentalement les conséquences de ce qu'il écrit sur l'exécution finale de son programme. Inversement, lorsqu'il parvient enfin à en faire accepter la syntaxe au compilateur, il n'est capable qu'au prix de lourds efforts de dire pourquoi l'exécution n'est pas conforme au résultat escompté.

Pourtant, comme le décrit Henry Lieberman dans la préface de [Lieberman 2001], une certaine vision la programmation, pour un néophyte, pourrait correspondre à celle que l'on se fait de l'enseignement. Un programmeur ne serait finalement qu'un professeur, capable d'apprendre certaines choses à une machine afin que celle-ci puisse les reproduire dans différents contextes. Or quoi de plus naturel pour un tel processus que d'utiliser des exemples ? Ainsi, la programmation pourrait consister à exécuter une instance du programme, pas à pas, pendant que l'ordinateur se chargerait d'enregistrer cet exemple au fur et à mesure de son déroulement. Et, même s'il faudrait sans doute mettre au point quelques techniques particulières pour indiquer, par exemple, ce qui varie d'une exécution à l'autre, il s'agirait là d'une approche naturelle de la conception de programmes informatiques.

Ainsi, la Programmation sur Exemple (PsE ou PbD pour Programming by Demonstration) repose sur un principe fondateur, évoqué en premier lieu par Dan Halbert ([Halbert 1984]) et repris par Allan Cypher ([Cypher 1993]) : « Le fait qu'un utilisateur soit capable d'exécuter une tâche dans un environnement donné devrait être suffisant pour que le système soit en mesure de créer un programme qui exécute cette tâche ».

Dans la suite de cette partie, nous recensons les différentes approches pouvant permettre à un utilisateur de personnaliser une application ou, mieux encore, d'en programmer de nouvelles. Grâce à ce tour d'horizon, nous verrons en quoi certaines techniques sont insuffisantes pour le but que nous cherchons à atteindre et comment certaines autres, très proches en apparence, se distinguent les unes des autres.

3.2 Personnalisation d'applications et programmation interactive

Encore à l'heure actuelle, la programmation reste une discipline obscure parce que mettant en jeu l'apprentissage de concepts abstraits et la manipulation de langages textuels à la syntaxe rébarbative. Dans des temps reculés de l'informatique, à une époque où les ordinateurs n'avaient pas encore fait leur entrée dans la plupart des foyers domestiques, les applications étaient commandées par des entreprises et donc réalisées en suivant un cahier des charges précis répondant, en principe, à des besoins spécifiques. De nos jours, les choses ont changé et, depuis l'apparition des ordinateurs personnels, les entreprises de logiciels créent des systèmes destinés à un très large public. Ainsi, le tableur Excel par exemple, est-il aussi bien utilisé par une ménagère pour faire la liste de ses courses que par une entreprise pour gérer l'état de ses stocks, ou encore par un biologiste pour exploiter le résultat de ses études. La contrepartie à cet état de fait est que l'utilisateur se retrouve souvent dans l'obligation d'effectuer des manipulations répétitives - et parfois complexes - pour parvenir à réaliser une action de base.

Cet aspect correspond à une des deux utilisations possibles de la programmation interactive : la personnalisation d'applications, c'est-à-dire la possibilité, pour un utilisateur dit final, de créer ses propres commandes à l'intérieur d'un environnement donné. La deuxième utilisation possible correspond à la création de programmes autonomes, déconnectés de tout environnement. Il nous paraît important de différencier ces deux facettes.

D'un côté il s'agit de programmation « dans l'interface de l'utilisateur » utilisant des commandes et des objets d'un domaine bien défini. Dans l'autre, il s'agit de programmation ayant recours à une aide graphique et pouvant être utilisée dans n'importe quel domaine. Nous verrons que l'approche que nous proposons peut être placée entre ces deux là : il s'agit en effet d'un environnement de programmation interactive travaillant sur un modèle générique capable d'englober différents modèles.

Dans cette partie, nous étudions les techniques existantes pour permettre à un utilisateur de personnaliser une application ou d'en créer de nouvelles sans avoir recours à la programmation textuelle classique.

3.2.1 Personnalisation d'applications

3.2.1.1 Préférences

Il s'agit sans doute de la méthode la plus simple et aussi la plus utilisée pour permettre à l'utilisateur d'un système de le faire coller le plus près possible à ses besoins. Les « préférences » se présentent la plupart du temps sous la forme d'une fenêtre de configuration (Figure 39).

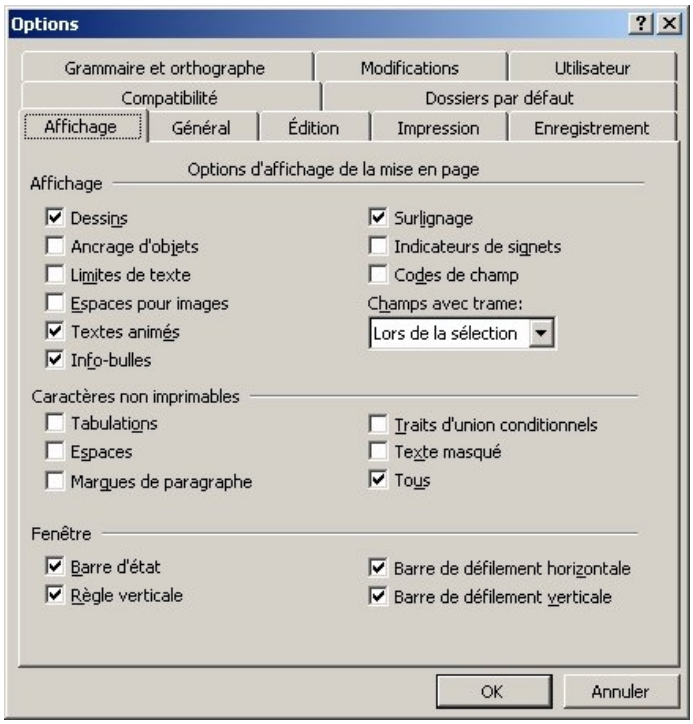


Figure 39 : Panneau de configuration de Microsoft™ Word 97

Une fenêtre de configuration offre ainsi, par l'intermédiaire de cases à cocher ou de boutons, un certain nombre d'options prédéfinies. Cela permet à l'utilisateur du système de modifier l'application pour la faire correspondre le mieux possible à ses besoins. Ceci étant, cela reste une alternative très limitée puisque les choix possibles sont déterminés à l'avance par le concepteur de l'application. De ce fait, les options disponibles restent très

généralistes. Il est par exemple impossible de prévoir, dans le cadre d'une application de traitement de texte, une opération qui change la couleur de tous les mots ayant une racine commune. D'autre part, cette approche est limitée par un problème de présentation : lorsque leur nombre augmente, les options de personnalisation deviennent rapidement difficiles à afficher. Un utilisateur renonce à trouver une option donnée à travers une forêt de panneaux …

Même s'il s'agit d'une solution facile à mettre en place, on ne peut pas considérer qu'il s'agisse de programmation et son caractère trop limité en bride l'utilisation. Ceci étant, cette solution est utilisée dans certains systèmes de simulation de métamorphoses. Par exemple, TOPL System [Terraz 1994] permet ainsi à un utilisateur de modifier interactivement des paramètres prédéfinis d'une animation modélisée par ailleurs, textuellement, grâce à un langage de programmation.

3.2.1.2 Langages de script

Une autre solution dans le cadre de la personnalisation d'application consiste à avoir recourt à un langage de script. Celui-ci se distingue des langages de programmation classiques car ils sont propres à un domaine d'application particulier (les objets et les actions sont spécifiques à l'application manipulée) et utilisent une syntaxe simplifiée a priori abordable par un utilisateur final. Par exemple, dans HyperCard, un programme Macintosh permettant à un utilisateur de créer des applications à manipulation directe, un langage de script nommé HyperScript donne la possibilité de décrire la position d'une boîte de dialogue par une phrase ressemblant à du langage naturel : « the location of the message box » (Figure 40).

```
on JumpToStack
      put the name of this stack into NameOfStack
      go to card 1 of stack Home
      doMenu « New Button »
      set the style of button « New Button » to roundRect
      put « on mouse up » & return into jumpScript
      put « go to » & NameOfStack & return after jumpScript
      put « en mouseUp » after jumpScript
      set the script of button « New Button » to jumpSCript
      set the name of button « New Button » to « Go To » & NameOfStack
      choose browse tool
end JumpToStack
```

Figure 40 : Un script HyperCard (tiré de [Cypher 1993])

Ceci étant, comme l'analyse Alan Cypher [Cypher 1993], les difficultés liées à la nature textuelle du script demeurent : même s'il s'agit d'un langage simplifié, il n'en reste pas moins un langage de programmation. La syntaxe, si elle ressemble à du langage naturel, est intolérante au moindre écart. Par exemple, « the position of the message box » n'est pas une phrase valide alors que « the location of the message box » l'est. D'autre part, la manipulation d'un tel langage nécessite la compréhension des concepts de programmation de base. Par exemple, quand on parle de « the message box », il faut que cela réfère à

un objet existant qui porte ce nom. On voit bien ici que « the message box » est une variable.

Même s'il s'agit d'une simplification intéressante, le langage de script reste d'un niveau d'abstraction trop élevé en regard aux concepts dont il demande l'appréhension (variables, boucles, etc.) pour être accessible à des utilisateurs finaux. D'autre part, il ne solutionne pas complètement le problème du caractère trop figé de la syntaxe.

3.2.1.3 Macros

Les macros constituent le premier véritable effort réalisé en matière de personnalisation d'application de manière totalement interactive. Il s'agit de permettre à l'utilisateur d'enregistrer une série d'actions réalisées interactivement. Celui déclenche manuellement le processus de mémorisation et effectue ses actions exactement comme il le ferait en temps normal. Le système enregistre pas à pas toutes les commandes utilisées, les mouvements de la souris, etc. Une fois son objectif atteint, l'utilisateur stoppe l'enregistrement. Il peut alors réutiliser la macro pour exécuter une nouvelle fois la série d'actions mémorisé.

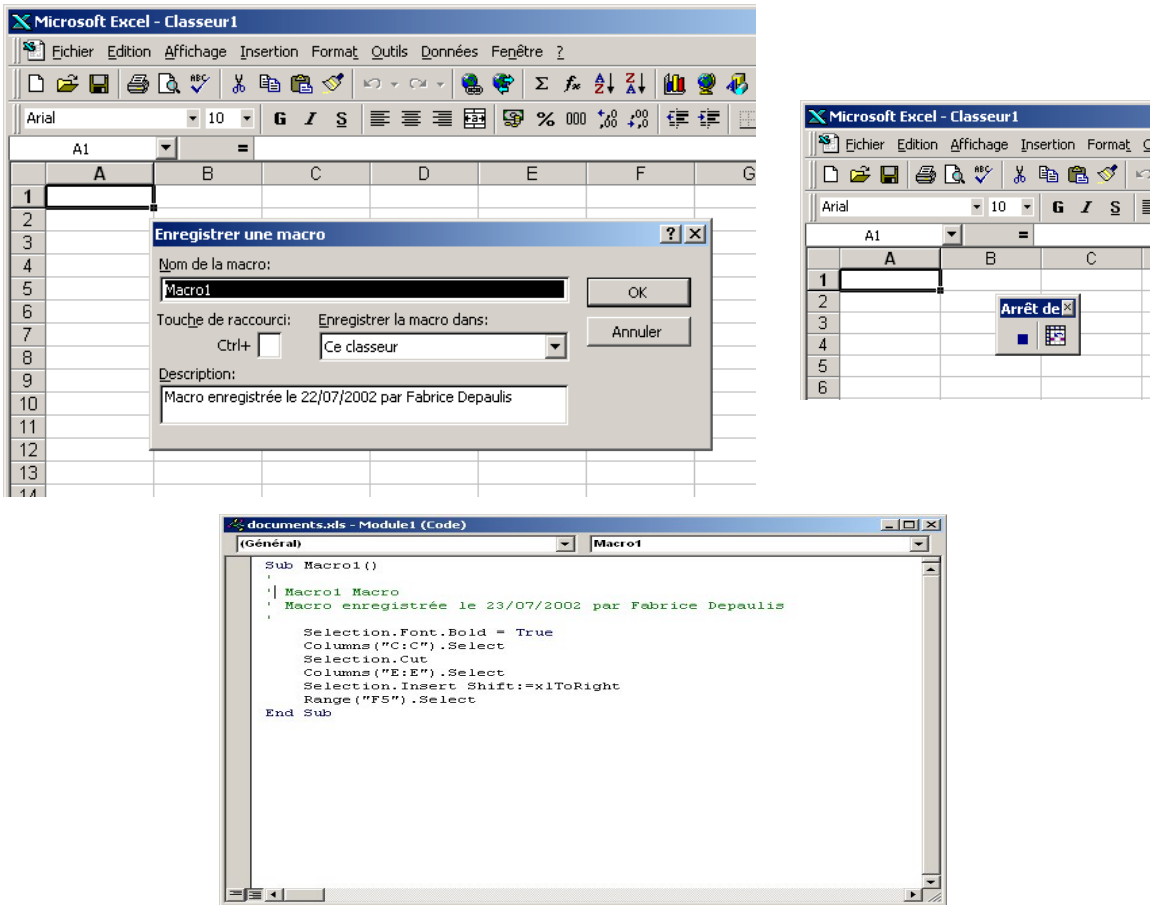


Figure 41 : Une macro sous Microsoft™ Excel

Les macros sont utilisées dans de nombreux tableurs, comme Microsoft™ Excel pour ne citer que lui. Ils sont particulièrement bien adaptés aux références précises qu'offrent ceux-ci dans le cadre d'un enregistrement d'action : par exemple, « déplacer le contenu de

la cellule A15 dans la cellule B12 ». Certains enregistreurs de macros se placent au dessus des applications et sont capables de reconnaître et d'enregistrer une séquence répétitive d'actions en se basant sur des événements comme l'appui sur une touche du clavier ou un déplacement de la souris. QuicKeys™ sur Macintosh™ par exemple est capable de mémoriser une action comme : « déplacer l'icône du fichier MonFichier sur le disque Sauvegarde ». Une telle action permet à un utilisateur d'automatiser la sauvegarde d'un ficher.

Le problème des enregistreurs de macros est qu'ils se placent à niveau d'abstraction trop bas, ce qui les empêche de généraliser les actions enregistrées. Si l'utilisateur d'un tableur veut déplacer la valeur d'un champ « nom » d'une cellule à une autre, l'enregistreur de macros va simplement mémoriser la référence de la cellule mais ne sera pas en mesure d'attacher une sémantique quelconque à la valeur déplacée. Du coup, si dans une autre feuille de calcul, le champ « nom » n'est pas placé exactement à la même position, l'exécution de la macro ne donnera pas le résultat escompté. De la même manière si le gérant d'une entreprise sélectionne tous ses employés, placés dans des cellules numérotées de B2 à B17, la macro doit être modifiée si un nouveau membre est embauché et qu'une nouvelle ligne est ajoutée.

Les macros constituent une solution envisageable pour des cas simples et systématiquement identiques, malheureusement, le niveau d'abstraction des actions enregistrées, trop bas, ainsi que l'absence de mécanisme de généralisation empêche la constitution d'un véritable programme informatique.

3.2.1.4 Conclusion partielle

Dans cette partie, nous avons étudié le fonctionnement de ce que l'on peut considérer comme étant les premiers efforts entrepris pour permettre à un utilisateur final sinon de créer des programmes, au moins d'en modifier quelques caractéristiques pour qu'ils correspondent de manière plus précise à ses besoins.

Malheureusement, que ce soient les panneaux de préférences, trop limités, les langages de script, puissants mais peu ou pas interactifs, ou les macros, générant des programmes trop simples, aucune des solutions étudiées ne permet d'envisager la création interactive d'algorithmes de la complexité requise pour la représentation d'une simulation de métamorphose. Comme nous l'avons vu dans la première partie de ce manuscrit, celle-ci correspond en effet à un véritable processus de programmation et simuler le comportement d'un phénomène naturel demande de manipuler toutes les notions propres à la programmation, que ce soit en terme d'objets (variables, constantes), qu'en terme d'exécution de programme (structures de contrôle).

Dans la partie suivante nous décrivons les difficultés majeures à surmonter pour écrire un programme, puis, en nous appuyant sur plusieurs taxinomies, nous montrons comment se distinguent les différentes approches dites de « programmation interactive » et pourquoi nous avons choisi d'utiliser la Programmation sur Exemple pour appuyer les applications de création de métamorphose.

3.2.2 Programmation interactive

Contrairement aux approches précédentes, ce que l'on qualifie de programmation interactive est le débouché de travaux visant à remplacer la programmation textuelle classique par des techniques graphiques et interactives. Il ne s'agit pas ici seulement de personnaliser ou de modifier par retouches superficielles des applications existantes, mais d'écrire de véritables programmes informatiques. Les différentes techniques de la programmation interactive vont de la programmation visuelle à la programmation sur exemple en passant par certaines méthodes issues de la recherche en CAO comme les systèmes paramétriques et variationnels.

Dans cette section, nous étudions dans un premier temps les difficultés posées par la programmation classique. Puis, à travers une taxinomie des systèmes de programmation, nous concluons sur la méthode la plus à même de remplir les besoins du domaine qui nous intéresse.

3.2.2.1 Difficultés de la programmation

Un programme est la description, dans un langage donné, sous une forme ou une autre [Girard 2000], d'un algorithme commandant un processeur. Guy Pierra [Pierra 1991] donne la définition suivante d'un algorithme :

> *« Etant donné une action abstraite à réaliser et un processeur définit par l'ensemble des objets qu'il sait manipuler et la liste de ses actions primitives, on appelle algorithme l'énoncé de l'ensemble 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 apparaître deux difficultés majeures de la programmation impérative :

- (1) Déterminer les variables de l'algorithme ;
- (2) Définir l'ordre d'exécution des actions primitives pour réaliser l'action abstraite de départ ;

Dans un programme, deux types d'objets sont manipulés. Les constantes ont une valeur qui ne varie pas d'une exécution à l'autre, les variables, elles, donnent au programme la possibilité d'être « différent » à chaque nouvelle exécution. Leur valeur peut être le fruit d'un calcul interne, on les appelle alors des variables internes, ou bien être fournies par l'environnement, il s'agit alors de paramètres. La première difficulté de la programmation consiste à identifier et à gérer ces deux types d'objet.

Le pouvoir d'expression d'un programme dépend de la possibilité de modifier le cours du programme et donc de la faculté à être en mesure d'utiliser les structures de contrôle principales que sont la séquence, l'alternative et la répétition ([Bohm & Jacopini 1966], [Mills 1975]) auxquels on peut ajouter le sous-programme. La seconde difficulté de la programmation telle qu'on l'a identifié précédemment consiste à intégrer les structures de contrôle non séquentielles à l'intérieur du programme.

Parmi les grands courants qui tentent de résoudre ces problèmes de manière interactive on peut citer la programmation visuelle, la programmation sur exemple, les systèmes paramétriques et variationnels. Voyons comment nous pouvons, en étudiant plusieurs classifications, comprendre comment les différentier.

3.2.2.2 Taxinomie des systèmes de programmation

La programmation visuelle, la programmation basée sur exemple et les systèmes paramétriques et variationnels constituent les approches principales utilisées pour créer des programmes informatiques interactivement. Malheureusement, du fait de leur objectif commun (créer des programmes) et des moyens mis en œuvre (l'utilisation de l'interactivité), il subsiste souvent une certaine confusion entre ces techniques. Nous présentons donc ici l'évolution des principales classifications de systèmes de programmation interactive afin d'en préciser nettement les différences.

En 1983, Schneiderman [Shneiderman 1983] introduit la notion de Manipulation Directe à l'usage de l'interaction graphique dans diverses tâches (programmation, accès aux bases de données, etc.), ce que MacDonald appelle déjà la Programmation Visuelle [Macdonald 1982]. Pour ces pionniers, l'utilisation d'images dans la programmation doit se généraliser et certains, comme Shu [Shu 1986], en analysent les raisons ainsi :

- En règle générale, et pas seulement en informatique, les gens préfèrent les illustrations au texte ;
- Une image est capable de véhiculer davantage de sens qu'un mot, voire qu'une phrase. Cela lui confère un caractère de concision que ne possèdent pas les signes littéraires ;
- Les images sont universelles, elles peuvent être appréhendées de la même manière par des personnes de nationalité, de culture ou d'âge différents.

Dès la fin des années 80, la multiplication des travaux en matière de langage de programmation graphique incite Chang à établir une première classification [Chang 1986]. Celle-ci repose sur deux critères :

- Les objets manipulés sont naturellement visualisables ou bien ils ne le sont pas. Dans le premier cas, leur représentation est évidente, dans le second il faut utiliser une schématisation du concept.
- Les structures du programme sont représentées de manière graphique ou textuelle.

Cette classification, si elle permet un premier regroupement, ne prend pas en compte les aspects dynamiques d'un programme. Par exemple, la représentation des variables et de l'évolution de leur état au cours de l'exécution du programme est une caractéristique importante d'un langage graphique.

Dans le même ordre d'idée, Shu [Shu 1986] distingue les « environnements visuels » permettant de représenter a posteriori tout ou partie d'un programme, des « langages vi-

suels » qui offrent la possibilité de construire interactivement tout ou partie d'un programme.

La lacune majeure que l'on peut voir dans ces classifications concerne le fait que, même si elles permettent de différencier les multiples systèmes graphiques, elles n'évaluent pas l'aide réelle en matière de programmation que peut apporter telle ou telle approche. Par exemple, est-ce que l'utilisateur peut manipuler des variables ? Si oui, alors de quelle façon le sont elles ? Textuellement ou graphiquement ? etc.

Pour ces raisons Halbert [Halbert 1984] le premier tente d'analyser le problème en prenant en compte la notion d'exemple. Pour lui, il est plus facile pour un utilisateur de comprendre le programme qu'il est en train de construire si celui-ci s'exécute en parallèle. Cette idée générale est reprise par Myers [Myers 1990] pour élaborer une nouvelle classification basée sur trois critères principaux :

- **Compilé/interprété.** Un système dit compilé voit toutes ses instructions traduites en langage de bas niveau avant leur exécution, ce qui nécessite des temps d'attente relativement longs entre la phase de commande et celle d'exécution. Au contraire, les instructions d'un système interprété sont traduites au fur et à mesure de leur création.
- **Programmation visuelle.** Pour Myers, un système peut se revendiquer d'appartenir à la classe des systèmes de programmation visuelle si l'utilisateur peut définir son programme « in a two – or more – dimensionnal fashion ». Cela exclut par exemple les langages textuels manipulant des données de manière graphique.
- **Programmation basée sur exemple.** Un système est dit « basé sur exemple » si une instance d'exécution se déroule parallèlement à la conception du programme.

Il adjoint un quatrième critère, secondaire, aux trois principaux :

- **Inférence.** Brad Myers reprend une distinction établie par Halbert [Halbert 1984] entre les systèmes sans inférence plausible (« Do What I Did ») et les systèmes avec inférence plausible (« Do What I Mean »). Ces derniers sont capables, selon la définition de l'auteur, de déduire des explications à partir d'informations limitées.

Dans [Girard & Pierra 1994], les auteurs reprennent cette classification et en montrent les insuffisances. En reprenant la définition d'un algorithme [Pierra 1991] et les difficultés inhérentes au processus programmation, ils établissent une nouvelle taxinomie en précisant notamment le rôle de l'exemple et la notion d'inférence, jugés trop flous dans celle de Myers. Pour les auteurs, il est indispensable de discerner les systèmes selon l'utilisation qu'ils font des « valeurs » d'un exemple. Celles-ci doivent avoir pour objet de « remplacer » par quelque chose de concret un ensemble de données abstraites. D'autre part, il faut distinguer à l'intérieur de ce que Myers appelle l'inférence (l'aptitude à générer de nouveaux faits à partir d'autres informations), le fait d'éviter à l'utilisateur une partie de la démarche de conception de l'algorithme, et le fait de lui permettre de transmettre facilement ce qu'il sait déjà lui-même de l'algorithme. Cette analyse débouche sur

une nouvelle catégorisation des systèmes de programmation, qui prend en compte les quatre critères suivants [Girard & Pierra 1994] :

- **Compilé ou interprété.** Comme dans la classification de Myers, on distingue ici les systèmes nécessitant un temps d'attente pour la traduction des commandes en un langage compréhensible par le système, de ceux dont l'interprétation est faite au fur et à mesure de l'exécution des actions ;
- **Programmation graphique ou non.** Un système de programmation est dit graphique si la définition du programme peut-être effectuée par interaction graphique dans un espace à deux dimensions ;
- **Basé sur exemple ou non.** Un système de programmation est basé 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 ;
- **Déclaratif ou impératif.** Un système de programmation est dit impératif si la structure de contrôle (autre que séquentielle) du programme construit peut être décrit explicitement par l'utilisateur. Si celle-ci est définie par le système à partir de relations entre objets définies par le programmeur, on parle de système de programmation déclaratif.

Cette dernière classification permet de distinguer nettement les différentes approches en matière de programmation interactive. Elle autorise à se livrer à une analyse des différentes propriétés que chaque technique apporte à l'utilisateur pour concevoir un programme et à mettre en évidence l'aide qu'elle est capable d'apporter en terme de programmation.

Prenons par exemple le système de programmation visuelle LabView™ (Laboratory Virtual Instrument Engineering Workbench) (Figure 42). Il s'agit d'un système type de cette catégorie : il est capable, grâce au langage G, d'aider un programmeur à concevoir des programmes de manière graphique et interactive. Dans ce type d'environnement de programmation, le concepteur doit être un véritable programmeur, le système n'apportant pas une abstraction des concepts mais « simplement » des outils graphiques de construction et de visualisation. Ainsi les variables et les structures de contrôle sont représentées de manière graphique, mais le concept que leur image véhicule reste le même pour l'utilisateur. Un tel environnement est catalogué, selon les critères de Patrick Girard, comme graphique, interprété, impératif et sans exemple.

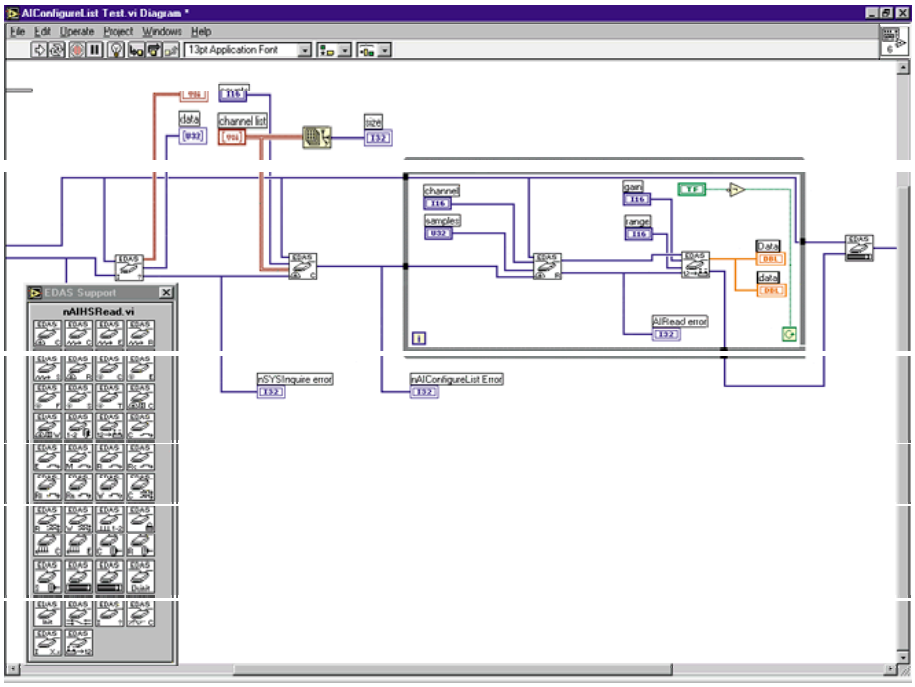


Figure 42 : L'environnement de programmation visuelle LabView

A l'inverse, EBP [Potier 1995] (Figure 43), un système de CAO permettant de créer des familles de composants, correspondant à de véritables programmes, abstrait certains concepts de la programmation, de façon à rendre celle-ci plus abordable. Ainsi, les variables ne sont elles jamais désignées comme telles : l'utilisateur référence uniquement des valeurs, prises sur son exemple courant, et le système interprète celles-ci comme autant de variables internes. Un tel système est classé dans la catégorie graphique, interprété, impératif et avec exemple.

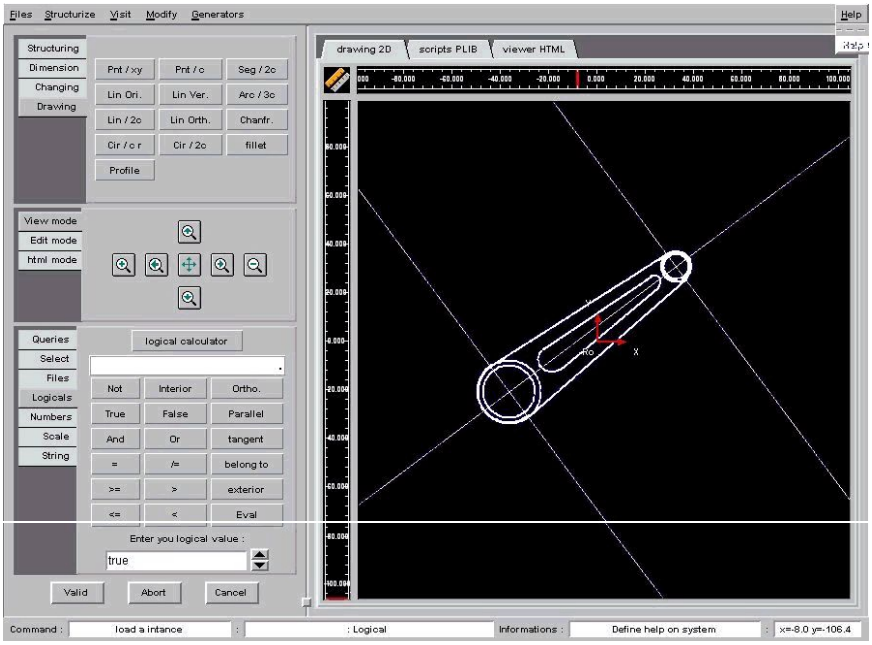


Figure 43 : EBP, un système de programmation sur exemple

3.2.2.3 Conclusion

Cette classification des langages interactifs nous permet de mettre en évidence les propriétés que nous voudrions voir appliquées à un système d'aide au développement de métamorphoses d'objets naturels. Comme nous l'avons déjà vu, la conception de telles transformations nécessite la mise en place de véritables programmes informatiques. Pourtant, la plus grande partie des utilisateurs potentiels de ces systèmes ne sont pas des programmeurs. Même s'ils sont capables d'en appréhender les concepts principaux, ils ne sont pour autant pas prêts à investir du temps dans l'apprentissage d'un langage spécifique (soit il graphique) ou d'une bibliothèque graphique particulière. D'autre part, il s'agit d'un domaine requérant une grande interactivité : le concepteur veut pouvoir visualiser le plus tôt possible dans le processus de développement les conséquences sur la métamorphose de l'ajout ou de la suppression de telle ou telle primitive.

Pour toutes ces raisons, la solution la plus adaptée nous apparaît être un système de programmation :

- **impératif**, le concepteur veut posséder un contrôle total sur la métamorphose ;
- **interprété**, le concepteur veut s'affranchir des temps de compilation alourdissant la phase de mise au point ;
- **graphique**, le concepteur n'est pas un programmeur et ne veut pas avoir à représenter textuellement un processus dont le résultat est purement graphique. La conception et la mise au point doivent pouvoir se faire dans le même mode de représentation que l'exécution ;
- **sur exemple**, le concepteur veut travailler sur une instance afin de réduire les efforts d'abstraction inévitable lorsque l'on travaille avec un langage de programmation « classique ».

L'étude menée sur la taxinomie des systèmes de programmation interactive nous permet de bien cerner le type d'environnement qui correspond le mieux aux attentes d'un concepteur de métamorphoses. La Programmation sur Exemple repose sur les concepts décrits ici et nous l'étudions donc de manière plus détaillée dans la partie suivante. Nous expliquons notamment les problèmes qu'elle pose et montrons les différentes solutions existantes pour les surmonter, sans pour autant essayer de les adapter à notre problème particulier. Ce dernier point fera en effet l'objet d'une section à part entière dans le dernier chapitre de ce manuscrit. L'idée est donc ici de sensibiliser le lecteur aux difficultés posées par la mise en œuvre de la Programmation sur Exemple et d'en exposer les solutions possibles.

3.3 Programmation sur exemple

L'idée de mettre la programmation à la portée de tous est sans doute à attribuer à Ivan Sutherland qui dès1963 [Sutherland 1963] développe le système Sketchpad et ébauche ce que la programmation interactive devrait être. Son travail aborde même la possibilité,

pour un utilisateur final, de modifier un système existant pour le faire correspondre à ses besoins spécifiques.

Mais c'est Halbert [Halbert 1984] le premier qui introduit la notion d'exemple en tant que véritable outil d'aide à la conception d'un programme. Il décrit en ces termes (repris par Cypher [Cypher 1993]) ce qui va devenir la programmation sur exemple :

> *« L'utilisateur écrit un programme qui effectue une tâche particulière en utilisant les mêmes commandes que celles qu'il utiliserait pour effectuer cette tâche de façon interactive. L'utilisateur programme dans l'interface du système. ».*

Tout ce que l'on cherche à faire en Programmation sur Exemple peut se résumer par ce principe fondateur.

3.3.1 Difficultés majeures

Les difficultés principales que l'on rencontre en programmation sur exemple sont les mêmes que celles de la programmation classique que nous avons déjà analysées précédemment en 3.2.2.1. Ainsi, à partir de la définition d'un algorithme donnée par Guy Pierra [Pierra 1991], on distingue deux problèmes majeurs :

- L'identification des objets manipulés : constantes, variables et paramètres ;
- L'introduction des structures de contrôle : séquence, alternative, répétition et sous-programme.

A ces problèmes de programmation classique viennent s'ajouter des difficultés particulières liées à la nature de la programmation dans les systèmes basés sur exemple. En effet, ceux-ci doivent permettre à un utilisateur final de créer des programmes de manière interactive, sans avoir à éditer des pages de code en mode textuel. En contrepartie, il faut trouver un moyen de présenter le programme à son concepteur dans une représentation qu'il soit capable de comprendre (celui-ci tient à pouvoir vérifier qu'il ne contient pas d'erreur par exemple). D'autre part, une fois cette représentation exhibée, on doit lui fournir la possibilité de modifier le code généré.

3.3.2 Identification et nomination

Le premier pas dans le passage du simple exemple d'exécution au programme proprement dit repose sur l'identification, à l'intérieur de cet exemple, des valeurs représentant respectivement les constantes, les variables et les paramètres. Plusieurs travaux évoquent l'importance de ce discernement ([Halbert 1984],[Olsen & Dance 1988],[Girard & Pierra 1990]). Pour que le résultat d'un programme ne soit pas systématiquement identique quelle que soit l'instance d'exécution, celui-ci doit comporter des objets dont la valeur est susceptible de changer, soit parce qu'elles sont le résultat d'un calcul (variables), soit parce qu'elles sont fournies par le contexte appelant (paramètres). De ce point de vue, la différence notable qui oppose un simple script (c'est-à-dire une suite d'actions et de va-

leurs constantes) à un véritable programme consiste en la présence, dans le second, de noms représentant chacun un ensemble de valeurs possibles.

En programmation textuelle « classique », lorsque le programmeur décide de créer un objet, il doit d'abord déclarer un identifiant, le plus souvent une chaîne de caractère, qui le représentera tout au long du programme. Il peut ensuite associer une valeur à cet identifiant et utiliser le nom pour faire référence à l'objet concerné. En programmation sur exemple, les choses sont différentes puisque la création de noms ne demande pas forcément l'intervention de l'utilisateur : par exemple, la création d'un objet graphique, puis sa désignation, toujours graphique, dans le déroulement d'une instance du programme ne doit pas nécessiter sa nomination explicite. Il s'agit d'une des caractéristiques de la programmation sur exemple qui permet à une variable de toujours avoir une valeur, y compris lors de la phase de construction, et autorise donc le concepteur à utiliser cette valeur pour référencer la variable. Ainsi, comme le décrit Jean-Claude Potier [Potier, et al. 1995], les noms de variables sont implicites et peuvent être générés par le système de manière transparente pour l'utilisateur.

La première étape de la généralisation d'un exemple pour créer un programme consiste à affecter des noms aux différentes valeurs fournies par l'utilisateur. Ceci étant, il faut également être capable de dire pour chacune d'entre elles s'il s'agit d'une constante, d'une variable ou d'un paramètre.

3.3.2.1 Paramètres

Les paramètres constituent un point essentiel dans la généralisation. Ils correspondent aux points d'entrée du programme et lui permettent de voir son déroulement varier d'une exécution à l'autre. Pour permettre d'identifier les paramètres et de les utiliser, on a l'habitude d'opposer une méthode **explicite** à une approche **implicite**.

Dans LIKE [Girard 1992], EBP [Potier 1995] ou TexAO [Texier 2000], trois systèmes de CAO dans lesquels la construction interactive de pièces peut être enregistrée sous forme de programme, l'utilisateur doit identifier seul, avant le début de la conception, les différents paramètres et leur donner une valeur ainsi qu'un nom (Figure 44). Il est essentiel que chaque paramètre possède une valeur afin que lors de la construction de l'exemple, la désignation d'un paramètre par son nom fournisse une valeur et que l'instance du processus de conception puisse continuer à se dérouler. Une fois cette étape réalisée, il peut construire un programme en faisant appel à tout moment à l'un de ces paramètres par son nom.

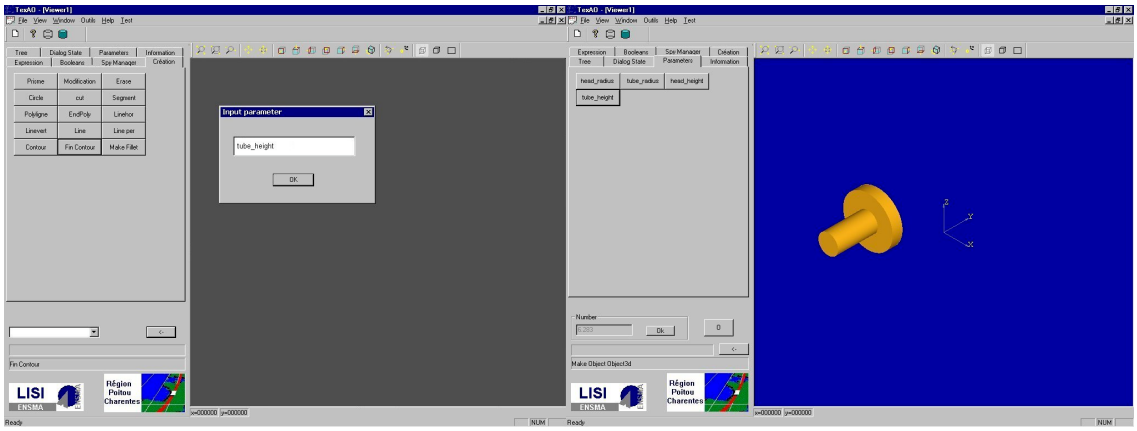


Figure 44 : Saisie et utilisation d'un paramètre dans TexAO [Texier 2000]

Dans [Bauer 1979], le système « Programming by Examples » propose une autre alternative, préconisée dans plusieurs autres systèmes de programmation sur exemple. On donne la possibilité à l'utilisateur de donner deux exemples d'exécution de son programme. Par comparaison, le système est alors en mesure de repérer les valeurs d'entrées qui ont différé et d'en déduire celles qui sont des paramètres.

De prime abord, cette seconde solution paraît plus séduisante pour un utilisateur non programmeur. Il semble qu'elle lui évite d'avoir à comprendre la notion de paramètre. Mais, en se penchant de plus près sur ce problème, on s'aperçoit que ce que Brad Myers appelle (à tort) dans ce cas de l'inférence, n'est en fait qu'une une simple convention de dialogue. En effet, afin d'exhiber deux exemples comprenant les informations nécessaires à pareille déduction, l'utilisateur doit faire preuve d'une projection mentale aussi complexe que celle consistant à désigner d'entrée les paramètres par des noms : il n'est pas affranchi de la nécessité de comprendre le concept même de paramètre. Pour cette raison, il nous semble préférable, dans le cadre d'un système tel que celui que nous voulons mettre en place, de laisser à l'utilisateur le soin de décider dès le début de l'enregistrement de son programme, les objets susceptibles de varier d'une exécution à l'autre. Ceci paraît un choix d'autant plus justifié, que le concepteur a conscience, dès le début de la conception des éléments qu'il désire voir varier d'un appel du programme à un autre.

3.3.2.2 Variables et constantes

Nous avons vu que la meilleure solution pour déclarer les paramètres d'un programme, dans le cadre de notre étude, était de demander explicitement à l'utilisateur de les nommer explicitement dès le début de l'enregistrement de son exemple. Ceci étant, on ne pourrait pas parler de programmation sur exemple si toutes les tâches de la programmation revenaient au concepteur. Or il est possible de donner la possibilité à un système de déduire des autres valeurs fournies par le concepteur les objets représentant des constantes et celles représentant des variables. D'autre part, il existe un mécanisme de nomination implicite qui soulage l'utilisateur de la nomination de ces variables : le contexte dynamique.

Dans un langage de programmation classique, le concepteur déclare explicitement tous les objets que le programme manipule. Le contexte ainsi constitué est placé en mémoire dès le début de l'exécution du programme et n'a pas a être modifié, il est dit « statique ». En programmation interactive, on veut pouvoir déclarer les variables de manière implicite : tout objet créé interactivement prend le statut de variable et doit pouvoir être référencé par simple désignation. Comme le programme se construit et s'exécute simultanément, la gestion du contexte n'est plus statique mais dynamique.

La gestion du contexte dynamique demande d'être capable de nommer automatiquement les variables (substitution nom/valeur) lors de la création du programme, mais également d'être en mesure d'effectuer la substitution inverse valeur/nom pour exécuter le programme. D'autre part, Jean-Claude Potier précise dans [Potier et al. 1995] la nécessité de vérifier la cohérence et la validité des objets manipulés.

La solution utilisée dans les systèmes LIKE [Girard 1992], EBP [Potier 1995] ou TexAO [Texier 2000] consiste donc à associer à chaque création d'objet la déclaration d'une variable nommée de manière automatique et ayant pour valeur cet objet. S'il ne s'agit pas d'un objet graphique, le système doit être capable de faire la distinction entre une constante et une variable. Pour cela, il suffit d'appliquer une règle simple, utilisée dans EBP [Potier 1995] :

- Si la valeur fournie par le concepteur met en jeu des valeurs d'autres objets, il s'agit d'une variable. On enregistre alors l'expression de son calcul ;
- Sinon, on considère que la valeur fournie est une constante.

D'après [Girard 1992] et [Van Emmerick 1991], la numérotation automatique des variables est la seule solution pour permettre leur déclaration de manière implicite. Ceci étant, cette numérotation automatique suppose que le même nombre d'objets soit construit d'une exécution à l'autre du programme. Par exemple, si les deux alternatives d'une condition ne comprennent pas la création du même nombre d'objets, que devient la signification, dans la suite du programme, de la référence à un objet qui n'a pas été créé ?

Dans la partie suivante, nous étudions la mise en place des structures de contrôle et nous précisons comment résoudre ce problème de nomination.

3.3.3 Structures de contrôle

Réaliser un programme dont le déroulement est purement séquentiel ne présente qu'un intérêt limité. En plus des variables et des paramètres, ce qui permet à l'exécution d'une application d'être différente d'une session à l'autre est la présence de structures de contrôle (alternative, boucle) propres à faire en sorte que certaines actions soient activées dans un certain contexte et pas dans un autre. La programmation textuelle classique propose deux grandes approches pour permettre la mise en place de ces concepts, l'approche **déclarative** (ou par contrainte) et l'approche **impérative**. Le problème qui se pose lorsque l'on travaille à un niveau interactif vient du fait que l'utilisateur, s'il est souvent capable

d'identifier les actions primitives à utiliser, n'est pas toujours compétent pour exhiber les structures de contrôle.

3.3.3.1 Approche déclarative

Cette première solution consiste à « cacher » le déroulement du programme au concepteur, un algorithme prédéfini (souvent appelé moteur) gérant l'exécution du programme. Le programmeur fournit des informations sur les objets manipulés ainsi que sur les éventuelles relations qui unissent ces objets. Celles-ci peuvent être exprimées à partir de règles (comme en Prolog II), de contraintes (Prolog III [Verroust 1990]) ou encore, comme c'est le cas dans certains tableurs, sous forme d'expressions arithmétiques. Une fois ces contraintes établies, le système s'occupe lui même d'établir l'ordre d'exécution des actions.

Dans les systèmes de programmation sur exemple basés sur cette approche, l'utilisateur peut créer des objets et définir de manière interactive des contraintes sur ces objets. Le moteur du système est chargé, lors de l'exécution, de faire en sorte que ces contraintes soient respectées à tout instant.

Cette approche a le défaut de sa qualité : le fait de rendre plus accessible la mise en place des structures de contrôle enlève une grande partie du contrôle que voudraient avoir certains utilisateurs sur le déroulement de leur programme.

3.3.3.2 Approche impérative

A l'inverse de l'approche déclarative, cette solution préfère donner au concepteur tous les outils lui permettant de décrire précisément l'enchaînement des actions de son programme. Les langages de programmation impérative manipulent ainsi explicitement le contexte du programme (variables et paramètres), les opérations primitives et les structures de contrôle : séquence, boucle, condition.

La mise en place interactive d'une telle solution n'est pas sans poser de problèmes.

3.3.3.2.1 Nomination

Nous avons vu précédemment que la nomination des variables pouvait être automatisée en utilisant ce que l'on a appelé le contexte dynamique. Un problème se pose cependant lorsque l'on se trouve en présence d'une structure de contrôle (alternative ou répétition). En effet, pour que la cohérence et la validité des objets créés soient sauves d'une exécution à l'autre, il faut que le nombre d'objets créés entre ces deux exécutions soit identique. Par exemple, si l'on considère un programme mettant en jeu une alternative. Celle-ci possède deux branches (une « Alors » et une « Sinon ») dans lesquelles le nombre d'objets créés est différent. La suite du programme subit alors un décalage au niveau de la numérotation susceptible d'entraîner des erreurs de références (Figure 45).

```
If (condition) then
       objet1 = creerObjet (ListeParametres1)
Else
       objet2 = creerObjet(ListeParametres2)
       objet3 = creerObjet(ListeParametres3)
Endif
objet4 = creerObjet(listeParametres4)
Affiche (objet3)
```

Figure 45 : exemple de décalage dans la numérotation automatique des variables

La solution utilisée dans les systèmes LIKE et EBP, et préconisée également par Loukipoudis dans [Loukipoudis 1996], consiste à gérer un contexte dynamique propre à chaque bloc de sous-programme à l'intérieur des structures de contrôle. Le contexte du programme, dans son intégralité, garde ainsi son intégrité.

3.3.3.2.2 Structures alternatives

Les blocs conditionnels posent un problème interactif. Etant donné qu'elles comprennent deux branches d'exécution distinctes, le système doit fournir au concepteur la possibilité de décrire les deux facettes de l'alternative. Une solution consiste à proposer à l'utilisateur de définir un premier exemple (correspondant à la branche ALORS) puis à ré-exécuter le programme jusqu'à l'évaluation du prédicat de la condition pour lui permettre de décrire un second exemple (correspondant à la branche SINON). La Figure 46 montre comment un menu permet à l'utilisateur de déclarer explicitement une structure conditionnelle dans EBP.

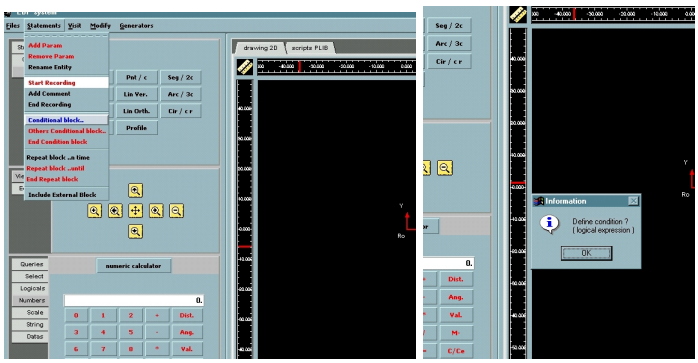


Figure 46 : Insertion explicite d'une condition dans EBP

3.3.3.2.3 Appréhension

La solution impérative permet de mettre à la disposition de l'utilisateur toute la puissance d'expression nécessaire pour qu'il puisse avoir un contrôle total sur l'application qu'il construit. Cette façon de voir les choses amène naturellement à se poser des questions sur l'utilisateur même. En effet, on ne peut pas demander à un enfant de comprendre les structures conditionnelles et de les manipuler avec la même habilité qu'un informaticien chevronné. Cela signifie que selon les compétences du public envisagé et selon la puis-

sance d'expression qu'il peut désirer, il faut adopter une solution impérative, déclarative ou encore mêler les deux approches pour obtenir un environnement adapté.

Dans le cadre de nos travaux sur un système d'aide au développement de métamorphoses d'objets naturels comme la croissance de plantes, nous considérons que l'utilisateur est un expert de son domaine mais pas de la programmation. Ceci étant, le fait qu'il ne soit pas informaticien ne doit pas nous empêcher de penser qu'il ne maîtrise aucun aspect de la programmation. Nous tentons de mettre en œuvre un système capable de se substituer complètement à la programmation textuelle de telles métamorphoses. Celui-ci doit donc être capable de proposer la même puissance d'expression, et les mêmes possibilités de contrôle. Mettre à la disposition de l'utilisateur des structures de contrôle ou lui demander de manipuler explicitement des paramètres ne doit pas être un problème car ce sont là des notions qu'il comprend au moins de manière intuitive s'il éprouve le besoin de les utiliser pour construire la croissance d'une plante.

La solution que nous avons retenue dans le cadre de nos travaux est donc l'approche impérative. L'environnement d'aide à la programmation de métamorphose, tel que nous l'avons envisagé, propose de déclarer explicitement les paramètres du programme ainsi que les structures de contrôles éventuellement utilisées.

La dernière section de cette partie présente une étude des problèmes posés par la représentation et la correction de programme dans les systèmes de Programmation sur Exemple.

3.3.4 Présentation et correction de code

La plupart des systèmes de Programmation sur Exemple s'intéressent à la création de programmes sans se préoccuper de leur visualisation [Depaulis 2000]. Pourtant, un utilisateur peut avoir de nombreuses raisons d'en consulter une représentation statique. D'après [Modugno & Myers 1993] l'utilisateur doit avoir accès à une représentation de son programme pour pouvoir en vérifier l'intégrité, le corriger, modifier les généralisations effectuées par le système ou encore le réutiliser dans le cadre d'un nouveau programme. Dans le cadre de la métamorphose, le concepteur veut, au même titre, être en mesure de comprendre pour quelle raison la simulation visualisée n'est pas celle qu'il attendait.

La représentation d'un programme créé par Programmation sur Exemple pose deux problèmes liés à la nature même de la PsE. Le premier concerne la structure du programme : la PsE évitant à l'utilisateur une partie de l'effort de conception, comment rendre un programme compréhensible à un non programmeur sachant qu'il n'en possède aucune représentation mentale ? Le second problème relevé par [Kurlander & Feiner 1993] est celui de la difficulté à représenter des objets qui ne sont pas convertibles de manière intuitive en termes textuels, comme par exemple des attributs graphiques ou des propriétés géométriques.

D'autre part, comme il est dit dans [Myers 1998], trop peu de systèmes de PsE s'intéressent à la correction, considérant que tous les programmes créés interactivement fonctionnent du premier coup et n'ont jamais à être modifiés. Ce problème est intimement lié à celui de la représentation : un utilisateur ne peut modifier un programme que s'il peut agir sur une représentation compréhensible de celui-ci.

La programmation de métamorphoses est un domaine dans lequel le concepteur a un grand besoin d'interactivité afin de pouvoir visualiser immédiatement sur la simulation, les modifications qu'il apporte à son programme. Il faut donc, autant que faire ce peut, être capable d'offrir à l'utilisateur un environnement dans lequel il puisse visualiser et corriger son programme dans un mode similaire à celui utilisé pour le définir.

Il est donc primordial de parvenir à offrir à l'utilisateur une représentation statique du programme exploitable, rendant compte à la fois de la structure et des objets manipulés. La littérature offre plusieurs exemples de solutions pour représenter et éventuellement corriger des programmes créés par PsE.

3.3.4.1 Textuelle

La plupart des systèmes se contentent d'une représentation textuelle des programmes qu'ils ont générés. Dans ces circonstances, les corrections se font presque toujours en éditant le code et en le modifiant « à la main ».

Tinker [Lieberman 1993b] est un environnement de programmation s'adressant à des programmeurs débutants et permettant de créer des programmes LISP. Pour construire un programme, l'utilisateur définit des instructions textuellement en désignant leurs arguments parmi la liste des instructions déjà connues. Chacune d'entre elles est composée d'une partie « Result » et d'une partie « Code », représentant respectivement l'instance de l'instruction et sa généralisation. Dans ce système la représentation du programme est textuelle et la correction se fait également textuellement, par ajout et suppression d'instructions.

Smallstar [Halbert 1993] permet d'automatiser des tâches répétitives dans un système d'exploitation graphique autorisant la manipulation directe. Pour enregistrer une action, comme par exemple déplacer le contenu d'un répertoire, l'utilisateur exécute simplement son opération interactivement. Il a ensuite accès à une représentation semi-textuelle semi-icônique du programme créé (les objets sont représentés sous la forme qu'ils ont dans le système d'exploitation mais les actions sont décrites de manière textuelle). Il peut utiliser cette représentation pour corriger le programme en modifiant « à la main » la généralisation (comme par exemple le critère de sélection du répertoire : nom, position, etc.) ou en ajoutant et supprimant des structures de contrôle.

Topaz [Myers 1998] est un système permettant de créer des programmes agissant sur les objets de différentes applications graphiques. Il permet aussi bien de subdiviser un rectangle en trois « sous-rectangles » dans un éditeur de dessin que de simplifier des associations de portes logiques dans un éditeur de circuits électroniques. La représentation d'un programme se réduit à la liste textuelle des actions qui le composent. Pour modifier

un programme, l'utilisateur peut sélectionner certaines actions pour les supprimer ou les dupliquer.

On voit bien à travers ces quelques exemples que de telles solutions sont difficilement acceptables : alors qu'on essaye d'alléger le travail du concepteur par des méthodes de construction toujours plus graphiques et interactives, on impose en même temps un formalisme de représentation rébarbatif et parfois simplement inaccessible. Dans le contexte qui nous intéresse, une telle représentation nuirait à l'interactivité que l'on désire mettre en place : le concepteur désire connaître l'aspect du programme qu'il crée mais au prix d'un effort de lecture limité.

3.3.4.2 Anticipation

Certains systèmes présentent la particularité de permettre aux utilisateurs de créer des programmes à leur insu. Ils partent du principe que, souvent, l'utilisateur ne sait pas que l'action qu'il va entreprendre est « automatisable » : la PsE est donc cachée et c'est le système qui repère lui-même les actions répétitives et tente de les automatiser.

Metamouse [Maulsby, Witten, & Kittlitz 1989] permet d'automatiser des tâches répétitives dans un éditeur graphique. L'utilisateur exécute des suites d'actions qui sont enregistrées et analysées par le système. Dès que ce dernier pense reconnaître un motif répétitif, il propose, par l'intermédiaire d'un agent « instructible » (la tortue Basil), d'exécuter le pas suivant du programme tout seul en montrant au concepteur ce qu'il pense être la prochaine action.

Eager [Cypher 1991] est également un système latent qui permet, lui, de repérer des tâches répétitives dans des applications de Bureautique. De la même manière que Metamouse, dès qu'il reconnaît un motif répétitif, il montre le prochain pas d'exécution en utilisant la surbrillance et propose à l'utilisateur de terminer l'action à sa place.

Dans ces deux cas, la représentation du programme est sous-jacente : le système n'en donne jamais une vue statique directement exploitable par l'utilisateur. Il se contente de lui donner un aperçu dynamique qui permet simplement de savoir s'il a bien saisi ses intentions. Cette solution, si elle convient dans un contexte où la programmation est totalement cachée à l'utilisateur, est impensable dans le contexte qui nous intéresse : on souhaite réaliser un système d'aide à la programmation de métamorphose dans lequel le concepteur sait qu'il est en train de créer un programme et veut pouvoir en consulter une représentation. D'autre part, de telles méthodes ne permettent pas d'envisager d'éventuelles corrections, alors que c'est justement un des aspects qui nous font envisager l'utilisation de la PsE.

3.3.4.3 Corriger sans représenter

Quelques systèmes tentent de résoudre le problème de la correction de programme sans le représenter de manière statique.

EBP [Potier 1995] est un système de CAO permettant de créer des familles de composants industriels à partir de la description interactive d'une seule de leurs instances. Il s'adresse à des experts en CAO. Pour créer le programme correspondant à la construction d'une pièce, l'utilisateur décrit les paramètres (hauteur, largeur, etc.) puis fabrique sa pièce interactivement. Il peut ensuite rejouer le scénario en donnant de nouvelles valeurs et obtenir ainsi une exécution différente (et donc une pièce différente). Dans EBP, il existe une représentation textuelle du programme créé, le système générant aussi bien des programmes Fortran que des fichiers d'échange Autocad ou EXPRESS. Mais cette forme n'existe que pour permettre des échanges entre systèmes et en aucun cas à des fins de visualisation. Il n'est d'ailleurs pas possible d'apporter des modifications au programme en éditant le code. Pour ça, EBP fournit un mécanisme interactif évolué consistant à exécuter le programme depuis le début jusqu'au point de modification et à enregistrer de nouvelles actions à partir de ce point.

Pavlov [Wolber 1996] est un système de création d'interfaces animées sans programmation. Il s'appuie sur un modèle Stimulus/Réponse : l'utilisateur spécifie le comportement de l'interface en associant interactivement à des actions de l'utilisateur les transformations graphiques que ceux-ci déclenchent. Aucune représentation statique du programme généré n'est disponible mais on peut accéder aux propriétés d'un objet de l'application et voir à quels stimuli ils répondent. Inversement, on peut dresser la liste des objets modifiés par un stimulus particulier. Cette représentation sous forme de feuille de propriétés ne permet de modifier le programme que textuellement.

Gamut [McDaniel 1999] est un système permettant de construire des animations et des jeux 2D. Le processus de construction d'un programme suit celui d'une perpétuelle correction : à tout moment, on peut modifier le comportement courant d'un objet de manière interactive. Deux boutons (Do Something et Stop That) permettent d'ajouter de nouvelles informations relatives au comportement d'un objet et d'annuler des comportements non désirés. Ainsi, si le système ne possède pas de représentation statique du programme, il est tout de même possible, sur une exécution, de le corriger de manière interactive.

Ces systèmes considèrent que l'utilisateur n'a pas besoin d'accéder à une représentation du programme autre que celle constituée par son exécution et mettent en place des moyens interactifs pour en permettre sa correction. Si ces solutions sont séduisantes pour des non programmeurs, elles ne peuvent convenir aux concepteurs de métamorphoses que dans une certaine limite : ceux-ci veulent pouvoir comprendre leur programme dans le détail et, si la forme dynamique suffit pour un certain nombre d'instructions simples, elle n'est pas satisfaisante pour comprendre des structures de contrôle. Le choix de la représentation dynamique est par contre totalement justifié dans des systèmes comme Pavlov et Gamut. Etant partiellement déclaratives, dans le sens où une partie non négligeable de la conception est laissée au système (déduction des structures de contrôle à partir d'informations limitées), ces applications n'ont pas à représenter des concepts que l'utilisateur n'est pas censé appréhender.

3.3.4.4 Comic-Strip Metaphor

La « Comic-Strip-Metaphor » est la seule solution graphique au problème de la représentation statique. Elle consiste, d'aprés [Kurlander & Feiner 1988], à représenter l'historique d'une application sous la forme d'une série de deux images, montrant l'état de l'application avant (prologue) et après (épilogue) l'exécution d'une action. Cette mise en forme permet de ne pas avoir à établir de nouvelles conventions, l'utilisateur faisant lui-même les opérations mentales menant d'un état à l'autre.

« History Based Macros by Example » [Kurlander & Feiner 1992] est une amélioration du système de macros de Chimera, un éditeur d'interfaces et de dessins en 2D. Il permet d'automatiser certaines tâches en utilisant un historique graphique des actions de l'utilisateur. Au contraire de nombreux systèmes, on peut, dans Chimera, décider de créer un programme après en avoir exécuté un exemple : Pour en concevoir un, l'utilisateur commence par effectuer une série d'actions, puis consulte l'historique graphique. Cet historique consiste en une séquence d'images portant le nom de chaque action et représentant l'objet affecté ainsi qu'une partie du contexte. Le concepteur peut ensuite sélectionner la série d'actions à partir de laquelle il veut créer un programme et déterminer les paramètres en attribuant des noms aux objets désignés comme tels. La représentation d'un programme est donc une partie d'historique graphique généralisée. Les modifications se font par édition de la séquence d'actions : on peut changer les paramètres ou ajouter de nouvelles transformations. Pour réaliser ce dernier point, toutes les actions postérieures à l'action insérée sont annulées puis réintroduites après la propagation de celle-ci.

Mondrian [Lieberman 1993a] est un éditeur graphique permettant de créer de nouvelles commandes par PbD. Dans ce système, toutes les commandes sont représentées par un doublon Prologue/Epilogue, montrant le type d'objet affecté et les effets produits. Pour décrire une nouvelle commande, l'utilisateur modifie un objet de départ en lui appliquant différentes opérations pour arriver à un objet final. Le programme consultable par l'utilisateur est composé des états successifs de l'objet, enregistrés sous une forme graphique. La commande associée est illustrée par un doublon utilisant la première et la dernière étape de la transformation. Contrairement aux macros de Chimera, on ne peut pas modifier le programme en utilisant sa représentation graphique.

KidSim [Cypher & Smith 1995], dont la dernière évolution n'est autre que l'application commerciale « Stagecast Creator™ », est un logiciel permettant à des enfants de créer des animations et des jeux 2D dans un espace discret. Les programmes sont gérés par des règles, décrites interactivement par l'utilisateur : il sélectionne un objet et son contexte et leur applique un certain nombre de transformations. La règle ainsi créée est consultable de manière graphique : elle comprend deux images représentant respectivement l'état de l'objet et de son contexte avant et après l'application des opérations de transformation. La suite des règles constitue le programme. Il n'est pas possible de modifier une règle autrement qu'en la décrivant à nouveau complètement.

Le problème de cette technique est toujours lié à la représentation des structures de contrôle. En effet, elle convient parfaitement à la visualisation étape par étape des effets des instructions du programme sur l'environnement graphique, et propose ainsi une représentation proche du mode de création du programme. Malheureusement, il est difficile de concevoir son utilisation dans le cadre d'un programme manipulant des structures de contrôle, car celles-ci sont, par nature, difficile à représenter de manière graphique.

3.3.4.5 Conclusion

La représentation et la correction de programme dans les systèmes de programmation sur exemple sont souvent des problèmes éludés. Alors qu'ils multiplient les méthodes interactives afin de permettre à l'utilisateur de s'abstraire de la syntaxe, ils ne proposent souvent qu'une représentation textuelle, difficilement compréhensible par l'utilisateur pour plusieurs raisons. D'abord parce qu'il s'agit souvent d'un formalisme qui lui est inconnu, ensuite parce que la création, dynamique, et la représentation, statique, diffèrent dans leur mode d'expression.

Ceci étant, nous avons vu un certain nombre de solutions qui pouvaient s'avérer satisfaisantes dans le cas de programmes simples : que ce soit la représentation dynamique ou encore les métaphores des « Comic Strip », il est possible de donner au concepteur une vue compréhensible des instructions simples du programme généré. Malheureusement, tous ces efforts sont réduits à néant dès que le système offre la possibilité de manipuler des structures de contrôle. La principale difficulté de représentation statique dans un mode graphique vient en effet de la présence de conditions et de boucles. Celles-ci manipulent, au niveau des conditions de contrôle, des expressions qui peuvent être complexes et dont la représentation graphique n'est pas toujours possible. Par exemple, la condition « Si l'âge de l'axe AX est supérieur à 5 années Alors … » n'est pas représentable graphiquement.

Dans les systèmes permettant la définition explicite de telles expressions, par opposition à ceux les générant à partir d'informations limitées, la conclusion qui s'impose est que les conditions doivent être représentées textuellement : un utilisateur qui définit une formule logique complexe de manière semi-graphique est capable d'en comprendre une représentation purement textuelle.

3.3.5 Conclusion

La Programmation sur Exemple poursuit le but de permettre à un utilisateur de créer des programmes de manière interactive. Malgré cet objectif commun, les systèmes existants diffèrent par les modes utilisés pour y parvenir : certains, intrusifs, tentent de déterminer si les actions d'un utilisateur sont répétitives afin de pouvoir les automatiser. D'autres prennent le parti de considérer que l'utilisateur sait pertinemment qu'il décrit un programme et apportent une aide appropriée. En fonction du public concerné, novice ou programmeur expérimenté, les applications de PsE offrent de surcroît des solutions

adaptées : génération automatique de certaines instructions (structures de contrôle) ou, au contraire, prise en compte explicite de tous les concepts de programmation complexes.

En dehors de ces choix fondamentaux, un certain nombre de problèmes, propres à la programmation, se posent de manière systématique : nomination des valeurs, déclarations des paramètres, distinction entre variables et constantes, expressions des structures de contrôle. D'autre part, la représentation et la correction du code, généré par le système, doit également faire l'objet d'une attention particulière, l'utilisateur n'étant pas forcément prêt à appréhender le programme sous une forme complètement déconnectée de celle utilisée pour sa description.

Une remarque générale que nous inspire cette étude de la PsE est que tous les systèmes existant sont construits pour un besoin spécifique : GAMUT pour créer des jeux en 2D, EBP pour construire des familles de pièces mécaniques, etc.

Nous verrons dans le chapitre suivant comment nous avons utilisé la Programmation sur Exemple pour permettre la création interactive de métamorphoses à partir d'un modèle générique, et comment nous avons résolu les problèmes que nous venons de voir dans ce contexte particulier.

4 Conclusion

Après avoir analysé les différentes caractéristiques et difficultés de la description de métamorphoses d'objets naturels structurés, nous nous sommes penchés, dans ce chapitre, sur les outils de conception nécessaires à la création d'une application permettant d'aider un utilisateur à créer des simulations de ce type de manière interactive.

Ainsi, nous avons vu que le cadre technique d'une telle application la faisait rentrer dans le domaine des AGICT. De ce fait, l'utilisation d'un dialogue structuré, permettant de décomposer un but complexe en un ensemble de sous-buts, plus faciles à atteindre, s'impose. Nous avons donc analysé les différentes architectures logicielles afin de savoir si l'une d'entre elle se prêtait mieux que les autres à cet exercice.

L'architecture H[4], basée sur ARCH, décrit de manière précise le fonctionnement d'un contrôleur de dialogue et permet de modéliser facilement un dialogue structuré. En se basant sur une décomposition des tâches du système en interacteurs de différents niveaux d'abstraction, un composant logiciel, le Moniteur, assure la transition des informations et l'indépendance des interacteurs entre eux. Ceux-ci sont ainsi capables de recevoir et de produire des données sans s'occuper de la provenance ou de la destination : ils peuvent de ce fait entrer dans l'expression de toute tâche complexe, sans qu'une interaction particulière soit prévue par le concepteur de l'application.

Dans un deuxième temps, nous nous sommes intéressé aux différentes techniques permettant de créer des programmes de manière interactive. A partir d'une classification des systèmes de programmation, nous avons vu que la Programmation sur Exemple était une solution correspondant à nos besoins : elle seule permet de créer interactivement des pro-

grammes manipulant les concepts analysés dans le premier chapitre de ce manuscrit. L'analyse des principes mis en jeu dans la PsE nous a permis de relever un certain nombre de difficultés et de disséquer les solutions proposées à ce sujet dans la littérature.

Le dernier chapitre de cette thèse se penche sur la création d'un environnement d'aide à la création d'applications de métamorphoses. Cet environnement générique peut être utilisé sur un modèle quelconque de représentation d'objet naturel structuré, et permet aux utilisateurs de l'application créée de bénéficier d'un moteur de Programmation sur Exemple. Celui-ci, se basant sur des propriétés imposées par le domaine, est capable de rejouer un scénario enregistré de manière interactive.

Chapitre 3

Un environnement de développement de simulation de métamorphoses

Résumé. *Le développement d'une application permettant la simulation de métamorphoses met en jeu un certain nombre de composants. Le caractère technique qui en fait une AGICT nous incite d'abord à l'utilisation du dialogue structuré, permettant à l'utilisateur de décomposer son action en buts et sous/buts. Or seule l'architecture H⁴ permet la modélisation simple d'un tel type de dialogue. Malheureusement, il semble de prime abord que la nature préfixée de celui-ci soit incompatible avec des types d'échange plus conviviaux comme la Manipulation Directe, intrinsèquement post-fixée. D'autre part, la programmation du contrôleur de dialogue lui-même revêt un aspect lourd et répétitif dès que la définition d'un grand nombre d'interacteurs et de questionnaires devient nécessaire. Nous montrons qu'il est possible d'intégrer des dialogues de type Manipulation Directe à l'intérieur de H⁴ en raffinant le rôle du moniteur et en utilisant des interacteurs spécifiques. Nous expliquons également comment nous avons allégé, grâce à un outil interactif, une partie de l'effort de codage du concepteur d'une application reposant sur H⁴.*
Une autre facette d'une application de création de métamorphoses concerne le caractère algorithmique du contrôle, qui contraint le système à mettre en jeu des techniques de programmation interactive, et plus précisément de Programmation sur Exemple. On peut décomposer les besoins en deux éléments que sont l'interpréteur de programme généré et le moteur d'enregistrement, capable de généraliser les actions de l'utilisateur pour créer le programme. Nous détaillons un tel interpréteur, basé sur le fonctionnement de H⁴, et nous expliquons les règles nécessaires à l'enregistrement et à la généralisation d'un scénario de croissance décrit de manière interactive. Cet environnement se fonde sur un modèle générique regroupant les caractéristiques des objets naturels structurés que nous explicitons également.

1 Introduction

Dans le premier chapitre de ce manuscrit, nous avons détaillé les techniques existantes en matière d'animation. A travers différents modèles et plusieurs systèmes, nous avons vu que seules les méthodes basées sur la topologie permettaient à un concepteur de simuler des transformations topologiques, en plus des modifications géométriques. Or celles-ci, s'appuyant en règle générale sur un système de contrôle algorithmique, posent un certain nombre de problèmes qui en brident l'utilisation. Le système niveau « animateur » contraint le concepteur à utiliser un langage de programmation pour décrire les différentes transformations de la simulation. Ainsi, celui-ci doit impérativement passer par des phases fastidieuses de « Codage – Compilation – Test – Correction » : il écrit d'abord le programme, le compile, le teste et éventuellement le corrige. Ce processus pose tous les problèmes inhérents à la programmation classique : il est long, il oblige le concepteur à maîtriser complètement un langage et tous les principes de programmation nécessaires à sa manipulation, et il le contraint à « jongler » entre deux représentations mentales de sa simulation, celle de la description textuelle et celle de l'exécution graphique. D'autre part, il s'agit d'un domaine où le concepteur raisonne le plus souvent en terme de « Modification – Test – Correction » et, dans ce cadre, le manque d'interactivité est d'autant plus insupportable.

Pour résoudre ces problèmes, nous avons étudié un certain nombre de techniques qui permettent à un utilisateur « final » de créer interactivement de véritables programmes informatiques, sans avoir pour autant à manipuler textuellement les concepts de la programmation. Parmi toutes celles-là, la Programmation sur Exemple apparaît comme la solution la plus adaptée aux difficultés rencontrées. Elle seule permet de créer de manière interactive des programmes comprenant tous les concepts algorithmiques nécessaires à la simulation de métamorphoses.

En nous basant sur l'étude d'exemples de croissances réalisée dans le premier chapitre et en reprenant les principes de la « Programmation sur Exemple » énoncés dans le deuxième, nous avons extrait un modèle générique sur lequel s'appuie un moteur de programmation sur exemple. En greffant son propre modèle, représentant un objet structuré, sur ce modèle générique, le concepteur peut mettre facilement à la disposition de ses utilisateurs des outils de construction et de « rejeu » interactifs de programmes de simulations de métamorphoses (Figure 47).

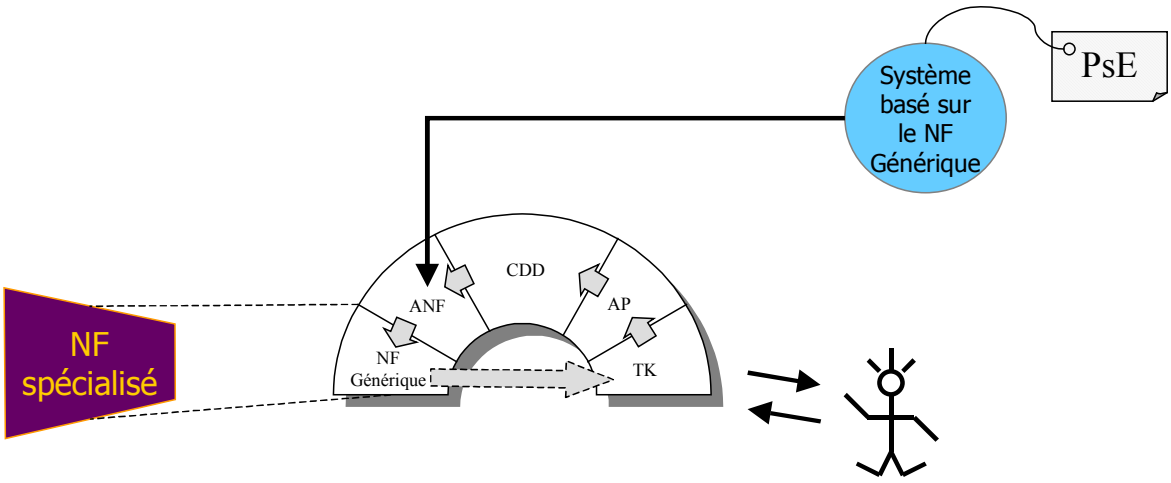


Figure 47 : Architecture de l'environnement de création d'applications de simulations de métamorphoses

Au cours de ce chapitre, nous montrons comment nous avons étendu et outillé l'architecture H⁴ de manière à la rendre utilisable dans le cadre d'une application interactive. Nous expliquons notamment comment, alors qu'elle impose un dialogue préfixé, celle-ci se trouve également en mesure de gérer un dialogue post-fixé, propre à celui de la manipulation directe.

Dans un deuxième temps, nous montrons en quoi notre approche diffère de celles existantes en matière de Programmation sur Exemple en répondant aux questions : qu'est-ce qui doit être lié au Noyau Fonctionnel ? Qu'est-ce qui doit être indépendant ? Qu'est-ce qui doit être du recours du concepteur de l'application et de celui de l'utilisateur ? Comment doit être représenté le programme construit ? Comment doivent être gérées les structures de contrôle ?

Nous détaillons ensuite les différents éléments de notre système, modèle générique et moteur de programmation sur exemple, ainsi que leur fonctionnement. Enfin, nous illustrons la mise en place de cet environnement sur une application de croissance de plantes en 1D½.

2 H⁴

2.1 Introduction

Comme nous l'avons déjà dit dans le deuxième chapitre, le dialogue structuré que propose H⁴ est une solution parfaitement adaptée au domaine technique auquel nous nous intéressons : l'utilisateur est en effet à même de construire des phrases d'interactions complexes correspondant à un besoin particulier, qui n'ont pas à être prévues à l'avance par le concepteur. Pour cette raison, nous avons choisi d'utiliser H⁴ pour l'architecture de notre environnement de développement.

Ceci étant, nous voulons mettre en œuvre des applications dans lesquelles l'utilisateur dispose des moyens interactifs les plus répandus pour dialoguer avec le système. Or, si H4 s'avère être un dispositif puissant pour gérer des dialogues structurés, elle ne semble a priori pas armée pour modéliser le dialogue « inverse », la manipulation directe. En effet, si l'on examine ces deux concepts du point de vue de la reconnaissance d'un langage, on s'aperçoit que le premier correspond à un langage préfixé (on donne la commande puis les opérandes) et le second à un langage post-fixé (on donne les opérandes puis seulement la commande). Ces deux modes, a priori antagonistes, peuvent cependant coexister et nous montrons dans cette partie comment H4 le permet.

Un autre problème posé par H4 concerne la difficulté de mise en œuvre du contrôleur de dialogue. La programmation de celui-ci s'avère souvent aussi répétitive que fastidieuse. Même si, dans l'implémentation que nous utilisons, les automates représentant les interacteurs sont générés de manière automatique, une grande partie du code est encore source d'erreur alors qu'il ne s'agit de que de spécifier les types manipulés et les questionnaires (des signatures de fonctions) à partir desquelles le système crée les automates. Toute cette partie peut être mécanisée et nous expliquons ici comment nous avons créé un outil graphique et interactif supportant cette charge.

2.2 H4 et Manipulation Directe

La manipulation directe est un style d'interface Homme-Machine rendu populaire par le Macintosh™, et qui est devenu un véritable standard d'interaction. Le terme a été employé pour la première fois par Shneiderman [Shneiderman 1983] pour décrire un style d'interface dans lequel l'utilisateur manipule des objets graphiques à l'écran par le biais d'actions physiques. L'impact des actions est immédiatement perceptible grâce à un retour (feedback) permanent à l'écran.

Dans le contexte d'applications de conception technique, dans lequel l'utilisateur veut posséder un contrôle total sur les processus de construction et de modification des objets, le dialogue avec le système doit autoriser la manipulation et l'utilisation des relations entre les différents objets. Le dialogue structuré de H4 est parfaitement adapté à ce cadre d'utilisation, puisque, comme nous l'avons vu précédemment, le contrôleur de dialogue est capable de reconnaître un langage préfixé. Ceci étant, même dans une application où le dialogue est a priori complexe, il subsiste toujours un certain nombre de tâches « simples », pour lesquelles un dialogue préfixé n'est pas adapté : déplacement de l'observateur dans une scène, zoom d'un élément, et même certaines modifications géométriques d'un objet. Pour cette raison, il est fondamental qu'une application interactive puisse profiter d'un mode « Manipulation Directe » permettant d'effectuer les tâches simples du système par des interactions adaptées. Le caractère antagoniste de ces deux logiques de dialogues (pré et post-fixés) est à l'origine des problèmes posés par l'intégration de la Manipulation Directe dans H4.

Dans la suite de cette partie, nous montrons comment introduire quelques uns des principes énoncés par Shneiderman [Shneiderman 1983] dans l'architecture H4. On veut ainsi pouvoir sélectionner un objet (par l'intermédiaire d'un pointé graphique) puis :

- Visualiser sa sélection par un retour graphique (dessin du contour, affichage de poignées) ;
- Le déplacer avec l'aide de la souris (en laissant le bouton de la souris appuyé) ;
- Modifier certains de ses attributs par l'intermédiaire de « poignées », manipulables à la souris ;
- Modifier certains de ses attributs par des opérations plus complexes (par exemple, modifier sa couleur en utilisant la combinaison des couleurs de plusieurs autres objets).

Nous verrons également que notre méthode peut-être étendue pour simuler d'autres comportements tels que la sélection de plusieurs objets ou la modification d'attributs par boîte de contrôle.

Au contraire de certains travaux déjà entrepris par le passé, et dont l'objectif était un contrôle particulier de l'application par l'intermédiaire de fichiers externes [Patry 1999], notre démarche modifie le rôle du moniteur et identifie clairement plusieurs interacteurs particuliers dédiés spécifiquement à la Manipulation Directe. D'autre part, nous replaçons ici notre réflexion dans le cadre de l'implémentation de H4 que nous avons utilisé.

2.2.1 Critères d'acceptation

Guillaume Patry [Patry 1999] met en avant les différents critères d'acceptation nécessaires selon lui à l'intégration de la Manipulation Directe dans un système interactif.

- **Transparence.** Le passage d'un mode à l'autre doit se faire de la façon la plus transparente possible. Pour que la Manipulation Directe garde tout son sens et donc sa simplicité, il est indispensable que l'utilisateur n'ait pas d'effort particulier à faire pour savoir s'il se trouve dans un mode de dialogue structuré ou dans un mode de manipulation directe. Le mode doit être prévisible par l'utilisateur selon le contexte.
- **Taux d'erreur.** De la même façon, la Manipulation Directe ne garde sa raison d'être que si elle n'ajoute pas de confusion dans l'esprit de l'utilisateur. Si l'adjonction de nouveaux modes de dialogue et de nouvelles fonctions augmente le risque que l'utilisateur se trompe, celle-ci perd tout son intérêt. Il faut donc éviter notamment que des équivoques ne se fassent dans l'expression d'une même commande dans deux modes coexistants. Par exemple admettons qu'il existe une commande « rotation » dans le mode de fonctionnement standard et que l'action correspondante soit également disponible en Manipulation Directe. Dans ce cas, le mode d'expression est différent : il s'agit bien de deux tâches différentes pour le dialogue, mais identiques pour l'utilisateur. Il est important que le comportement

du système soit prévisible lorsque l'utilisateur « mélange » les deux modes d'expression d'une même tâche.

- **Développeurs.** L'intégration de la Manipulation Directe ne doit pas entraîner une surcharge de travail trop considérable pour le concepteur de l'application. Pour qu'elle soit utilisable, son développement ne doit pas demander d'effort de conception trop lourd. D'autre part, elle doit respecter une certaine homogénéité avec le reste du système.

Ces critères nous ont servi de ligne directrice dans la mise en place de notre méthode.

2.2.2 *Comportements*

Pour mettre en place la Manipulation Directe, il nous faut analyser le fonctionnement des principales actions qu'offre ce mode de dialogue. Les travaux de [Olsen 1998], analysés notamment dans [Patry 1999], étudient les différents comportements de Manipulation Directe que sont la « Sélection et Translation » et « l'Utilisation de Poignées » d'un objet. Nous reprenons cette analyse afin de spécifier clairement les points essentiels nécessaires à l'intégration de ces comportements.

2.2.2.1 Sélection/Translation

La Sélection/Translation est l'action de base que l'on souhaite pouvoir utiliser dans un système interactif moderne, manipulant différents objets graphiques. Elle consiste à donner à l'utilisateur la possibilité de sélectionner un objet avec la souris (par simple clic), puis à le mouvoir, en déplaçant la souris sans en lâcher le bouton. Ce mouvement provoque généralement l'apparition d'un « écho fantôme », représentant l'objet grisé, qui suit le curseur de la souris. Lorsque l'utilisateur relâche le bouton, l'écho temporaire disparaît et l'objet est réellement déplacé (Figure 48).

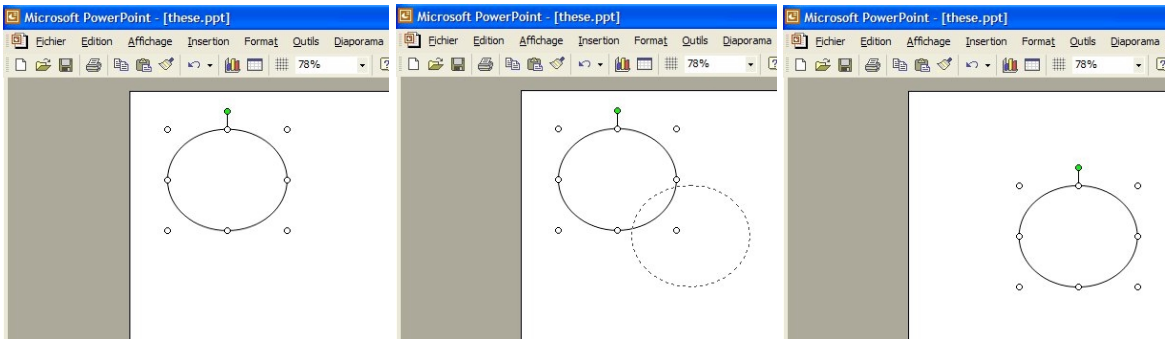


Figure 48 : Sélection - Translation dans Microsoft™ Power Point

Le comportement du contrôleur de dialogue d'une application pour la gestion d'une Sélection/Translation peut se résumer ainsi [Patry 1999] (on se limite à un objet unique) :

- A partir d'un état initial, si le contrôleur reçoit un clic bas et que les coordonnées de ce clic correspondent à un objet graphique, celui-ci est sélectionné pendant

que les autres objets sont désélectionnés. Le système passe dans un état « Sélection ».

- A partir de cet état « Sélection », lorsque l'utilisateur relâche la souris, le contrôle revient dans un état initial.
- Toujours à partir de cet état « Sélection », si l'utilisateur déplace suffisamment la souris, le système commence à déplacer l'objet sélectionné sous forme de fantôme. Il passe alors dans un état « Déplacement ».
- Dans l'état « Déplacement », la réception d'un événement de relâchement du bouton de souris entraîne le déplacement effectif de l'objet sélectionné, sous forme d'une translation de vecteur. La réception d'un mouvement de souris entraîne quant à lui le déplacement du fantôme de l'objet sélectionné, sans que le système ne change d'état.

2.2.2.2 Poignées

Les poignées sont des points de contrôle associés à un objet graphique (Figure 49). Elles permettent d'effectuer des opérations complexes en règle générale de nature géométrique, comme la rotation ou l'homothétie.

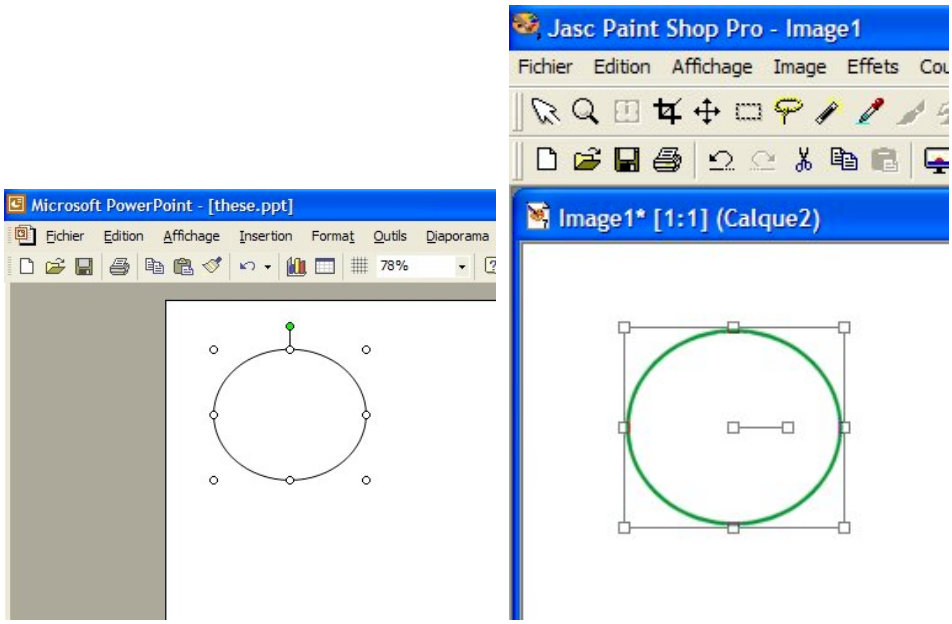


Figure 49 : Représentations de poignées dans Microsoft™ Power Point et PaintShopPro™

Il est important de noter que seuls les objets eux-mêmes sont en mesure de créer les poignées, car eux seuls connaissent leur géométrie. Ils sont donc nécessairement responsables du placement des poignées qui leur sont associées. D'autre part, seuls les objets, encore une fois, peuvent connaître les actions associées à une poignée donnée : en effet, une poignée peut être chargée de la rotation de l'objet pendant qu'une autre est utilisée pour appliquer une homothétie, etc.

Les poignées sont affichées et deviennent manipulables dès que l'objet est sélectionné. C'est le contrôleur de dialogue qui est chargé de détecter le fait qu'une action a lieu sur

une poignée : à partir de l'état initial, si le système reçoit un clic bas à proximité d'une poignée, le dialogue passe dans un état « Poignée ». A partir de là, la réception d'un déplacement de souris entraîne la transmission des informations de déplacement (le dialogue ne change pas d'état). La réception d'un clic haut met fin à la manipulation de la poignée.

2.2.3 Approche globale

Le contrôleur de dialogue d'une application à manipulation directe est généralement implémenté au sein de la fenêtre de visualisation [Olsen 1998]. De là il reçoit les événements qui lui sont destinés et peut analyser les entrées de l'utilisateur comme on vient de le décrire dans les sections précédentes. Cependant, une telle démarche n'est pas possible dans le cadre d'un dialogue structuré : les événements sont passés à une tâche qui les attend et qui va pouvoir, à son tour, transmettre un résultat à une autre tâche (c'est le rôle du moniteur et des interacteurs).

Pour intégrer ces différents comportements dans H⁴, notre approche consiste à décomposer les tâches en plusieurs composants, correspondant à autant d'interacteurs. Les tâches à réaliser sont :

- Sélection d'un objet à partir d'un « clic bas » ;
- Sélection d'une poignée à partir d'un « clic bas » ;
- Déplacement d'un objet selon les positions successives données par des « mouse move » ;
- Modification des propriétés géométriques de l'objet en fonction de la poignée sélectionnée et du déplacement de la souris.

Rappel sur H⁴

Avant de détailler la solution que nous avons choisie, il est bon de rappeler ici les principales caractéristiques de l'architecture H⁴ et plus précisément du fonctionnement de son Contrôleur de Dialogue.

Comme nous l'avons déjà expliqué en 2.2.5, le contrôleur de dialogue est constitué d'un ensemble de composants logiciels.

- Les jetons sont une abstraction des données de la Présentation (clic, commande de menu, etc.) et de celles du Noyau Fonctionnel.
- Les questionnaires représentent les tâches du système. Ils sont défini par leur nom et leurs paramètres d'entrée et de sortie et possèdent un lien vers une méthode de l'Adaptateur de Noyau Fonctionnel.
- Les questionnaires sont regroupés au sein d'interacteurs de même niveau d'abstraction (les tâches de création par exemple font partie du même interacteur). Ils sont représentés par des automates dont les transitions sont les paramètres des différents questionnaires.

- Le Moniteur organise hiérarchiquement les différents interacteurs. C'est lui qui reçoit les jetons venant de la couche de Présentation et qui les transmet aux différents interacteurs, dans l'ordre de la hiérarchie.

Le fonctionnement est alors très simple. Lorsque le Moniteur reçoit un jeton, il le transmet au premier interacteur dans la hiérarchie. Celui-ci accepte le jeton s'il correspond à une transition à partir de son état courant et le stocke dans un registre. Si aucune transition ne correspond à ce jeton, il le refuse. Dans ce cas, le Moniteur propose de nouveau le jeton à l'interacteur suivant dans la hiérarchie jusqu'à ce qu'il soit utilisé ou que l'on arrive au dernier interacteur. Dans le cas où un interacteur a reçu tous les jetons nécessaires à l'activation d'un questionnaire, celui-ci appelle la méthode de l'ANF correspondante. Les données sont alors transformées en données du Noyau Fonctionnel et la primitive du Noyau Fonctionnel est à son tour appelée, rendant effective l'action de l'utilisateur. Si cette méthode produit une valeur, celle-ci est transformée en jeton et rendue au Moniteur, lequel le propose à l'interacteur suivant dans la hiérarchie.

Ce rappel nous permet de bien insister sur l'importance du placement des interacteurs : si un interacteur produit un jeton « utilisable » uniquement par le questionnaire d'un interacteur placé EN DESSOUS dans la hiérarchie, ce dernier ne sera jamais en mesure de le recevoir.

Principe de fonctionnement

Voyons maintenant l'idée globale sur laquelle repose la modélisation de la Manipulation Directe dans H[4].

Lorsque l'utilisateur se trouve dans un mode de manipulation directe et qu'il utilise la souris pour cliquer sur l'écran, un jeton de type « clic bas » est envoyé au Moniteur. Celui-ci le transmet à un interacteur chargé de la sélection des objets en manipulation directe. Deux cas peuvent alors se produire :

- Le pointé fourni se trouve à proximité d'un objet. L'interacteur retourne alors l'objet en question sous forme de jeton, lequel est transmis par le moniteur à un nouvel interacteur chargé de gérer les interactions par manipulation directe sur l'objet sélectionné. Cet interacteur est activé lorsqu'il reçoit un jeton « objet » et peut être configuré pour effectuer des actions selon les jetons (« mouse move », « clic haut », « poignée » notamment) reçus par la suite.
- Le pointé fourni ne se trouve pas à distance suffisante d'un objet. L'interacteur retourne le jeton « clic bas » qu'il a reçu au moniteur, lequel le transmet à l'interacteur de manipulation directe.

Cette situation exclut la gestion des poignées. Leur sélection est gérée de la même manière. Lorsque l'interacteur de sélection d'une poignée reçoit un « clic bas », deux cas peuvent survenir :

- Le pointé fourni se trouve à proximité d'une poignée. L'interacteur retourne alors la poignée en question sous forme de jeton, lequel est transmis par le moniteur à un nouvel interacteur chargé de gérer les interactions par manipulation di-

recte sur un objet. Si cet interacteur est bien en attente d'un jeton de ce type, celui-ci est consommé, l'action associée exécutée.

- Le pointé fourni ne se trouve pas à distance suffisante d'une poignée. L'interacteur retourne le jeton « clic bas » qu'il a reçu au moniteur, lequel le transmet à l'interacteur de manipulation directe.

Ce mode de fonctionnement met en évidence plusieurs éléments de réflexion. Tout d'abord, une implémentation de H[4] prévoyant un certain nombre de jetons par défaut se contente généralement de définir le jeton correspondant au pointé graphique, sans différencier les clics haut, des clics bas ou encore prendre en compte les mouvements éventuels de la souris. Il s'agit dans notre cas de jetons fondamentaux qu'il faut pouvoir manipuler au niveau du contrôleur de dialogue pour décrire un dialogue de manipulation directe.

D'autre part, on voit bien ici que les interacteurs que l'on voudrait utiliser suivent une logique différente de celle des interacteurs « classiques ». En effet, alors que ceux-ci reposent sur un questionnaire du type :

```
[Nom_Questionnaire, « commande », « jeton 1 », « jeton 2 »,..., « jeton N »]
```

les interacteurs dédiés à la manipulation directe répondent à un questionnaire du type :

```
[Nom_Questionnaire, « jeton 1 », « jeton 2 »,..., « jeton N »]
```

On s'aperçoit également, avec cette description, qu'il faut se poser un certain nombre de questions sur le placement relatif et absolu des trois interacteurs énoncés. Par exemple, si l'on place l'interacteur de sélection d'un objet par manipulation directe SOUS l'interacteur de sélection d'une poignée, il sera alors impossible de sélectionner une poignée : le clic bas fourni sera d'abord récupéré par l'interacteur de sélection d'objet qui rendra l'objet correspondant au Moniteur, lequel ne pourra pas proposer de jeton « clic bas » à l'interacteur de sélection de poignée (si on considère bien entendu que la poignée d'un objet est toujours placée dans la zone de sélection de l'objet auquel elle appartient). D'autre part, où faut-il placer ces interacteurs pour que l'on puisse les utiliser dans certaines phases d'un dialogue structuré ? Par exemple, pour pouvoir faire tourner un objet sur lui-même afin de rendre un de ces attributs plus accessible, sans pour autant perturber la « phrase » de construction d'un objet en cours.

Enfin, cette description informelle du fonctionnement général que l'on voudrait voir suivre à notre architecture repose sur une supposition : « si le dialogue se trouve dans un mode de manipulation directe alors … ». Or il manque à l'architecture un moyen de connaître le mode de dialogue dans lequel se trouve le système à un moment donné.

Dans la suite de cette partie, nous expliquons comment résoudre ces différents problèmes et nous décrivons précisément les différents interacteurs ainsi que le rôle du moniteur dans ce contexte particulier.

2.2.4 Solution détaillée

La solution que nous avons retenue pour permettre la modélisation d'un dialogue de type Manipulation Directe dans H4 repose sur l'utilisation de jetons spécifiques ainsi que la spécification d'interacteurs spécialisés et la modification du rôle du moniteur.

2.2.4.1 Jetons

Dans H4, le concepteur de l'application crée les jetons dont il a besoin. Chaque objet du Noyau Fonctionnel ou de la Présentation devant transiter par le Contrôleur de Dialogue doit ainsi avoir une représentation sous forme de jeton. Ceci étant, dans une implémentation donnée de H4, un certain nombre d'entre eux sont définis « par défaut », car présents dans la majorité des applications. C'est le cas du jeton représentant les pointés graphiques ou de celui représentant les entiers. L'intégration de la Manipulation Directe sous la forme d'interacteurs requiert l'introduction de nouveaux types de jetons « par défaut », relatifs aux différents événements de la souris que l'on veut prendre en compte (clic bas (bouton appuyé), clic haut (bouton relâché), mouse move (déplacement de la souris)) ainsi que ceux correspondant aux poignées graphiques.

- Jeton « M↓ » correspondant à un clic down
- Jeton « M~ » correspondant à un mouse move
- Jeton « M↑ » correspondant à un clic up
- Jeton Poignée comportant des informations de position et de type (on peut avoir plusieurs types de poignées)

2.2.4.2 Interacteurs de manipulation directe

La modélisation de ces tâches sous forme d'interacteurs pose un problème. Alors que dans H4 les interacteurs sont activés par des commandes explicitement fournies par l'utilisateur (dialogue préfixé), l'expression des actions est donnée, dans le cadre d'un dialogue de manipulation directe, sous forme post-fixée (l'utilisateur fournit d'abord l'objet sur lequel va porter l'action). Or il faut bien comprendre que l'on ne peut pas impunément modifier la nature des questionnaires des interacteurs sans modifier profondément le comportement de H4. Pour que l'architecture conserve toutes ces propriétés, il faut notamment conserver la structure des interacteurs et donc des questionnaires qui les définissent. Pour cette raison, il est nécessaire de montrer que les interacteurs de manipulation directe peuvent se rapporter à des interacteurs classiques, c'est à dire qu'une signature de ce type :

```
[Nom_Questionnaire, « commande », « jeton 1 », « jeton 2 »,..., « jeton N »]
```

peut se ramener à une autre du genre :

```
[Nom_Questionnaire, « jeton 1 », « jeton 2 »,..., « jeton N »]
```

Un questionnaire tel que celui-ci permet à un interacteur d'être activé autrement que par la réception d'un jeton commande. Ceci est indispensable en Manipulation Directe car les actions sont commandées de manière implicite (et non au moyen d'un bouton ou d'un

menu). Un interacteur dont les questionnaires sont basés sur ce format peut être simulé à partir de la définition initiale : il suffit de considérer que la réception d'un « clic bas » est une commande (seuls les « clics bas » initient les phases de Manipulation Directe).

A partir de cette nouvelle définition, nous pouvons répartir les différentes tâches dans trois interacteurs différents.

2.2.4.2.1 Sélection d'un objet

Un interacteur est chargé de la sélection d'un objet en manipulation directe. Il reçoit un jeton « clic bas » et rend un jeton contenant l'objet le plus proche de ce clic, s'il existe, ou bien le même jeton « clic bas » dans le cas contraire (Figure 50). La transmission à l'identique du jeton reçu dans le cas où aucun objet ne se trouve à proximité du pointé est nécessaire pour que la position fournie soit éventuellement réutilisée par un interacteur de niveau supérieur.

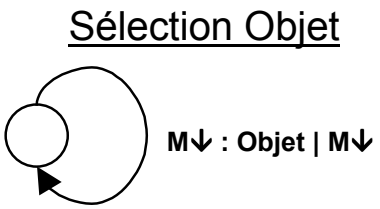


Figure 50 : Automate représentant l'interacteur de sélection d'un objet en manipulation directe. Celui ci possède une unique transition sur un « Clic Bas » et rend ou bien un Objet ou bien le même « Clic Bas ».

Remarque. *Les interacteurs de manipulation directe et notamment de sélection (Poignée, Objet) sont toujours actifs : tout se passe comme s'ils n'étaient pas activés par une commande. Leur existence n'empêche pas celle d'un interacteur de sélection « non manipulation directe ». Ils peuvent cohabiter car ils n'ont pas la même signature. Par exemple, l'interacteur de sélection d'un objet en mode « non manipulation directe » possède une transition sur un « pointé » et non sur un « clic bas ».*

2.2.4.2.2 Sélection d'une poignée

Comme nous le verrons dans le paragraphe suivant, pour des raisons de transmission de jetons, c'est un autre interacteur qui est chargé de la sélection d'une poignée en manipulation directe. Il reçoit un jeton « clic bas » et rend un jeton contenant la poignée la plus proche, si elle existe, ou bien le même jeton « clic bas » dans le cas contraire (Figure 51). La transmission à l'identique du jeton reçu dans le cas où aucun objet ne se trouve à proximité du pointé est nécessaire pour que la position fournie soit éventuellement réutilisée par un interacteur de niveau supérieur.

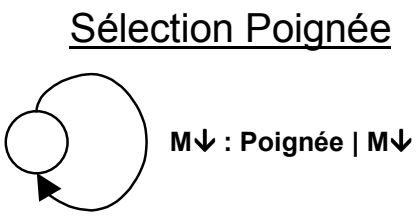


Figure 51 : Automate représentant l'interacteur de sélection d'une poignée en manipulation directe. Celui ci possède une unique transition sur un « Clic Bas » et rend ou bien une Poignée ou bien le même « Clic Bas ».

2.2.4.2.3 Interacteur de Manipulation Directe

Un dernier interacteur est chargé de représenter le dialogue de la manipulation directe sur un objet donné (il est le véritable interacteur de manipulation directe). Quand il reçoit un objet, il passe de son état initial à un état « sélectionné », dans lequel il attend une série de jetons qui va lui permettre d'activer une des fonctions liées aux différents questionnaires qui le composent. Toutes les opérations de manipulation directe sont possibles : translation, modification par poignées, modification par un ensemble de paramètres, il suffit qu'un questionnaire prévoyant la série de jetons à récupérer ait été spécifié (Figure 53). Il faut bien noter ici que l'interacteur est capable d'activer une action portant sur une série de jetons en cours. Par exemple, si l'on a reçu un objet et un déplacement de souris, on peut appeler une primitive qui déplace l'objet selon la position donnée par le déplacement. A partir de là, tous les comportements de la Manipulation Directe sont possibles. Le concepteur doit simplement spécifier les actions à accomplir en fonction des jetons reçus. Ainsi, simuler la translation de l'objet sélectionner revient simplement à ajouter un questionnaire du type de celui de la Figure 1 en spécifiant une action de translation sur le « Mouse Move » et une action de déplacement définitif de l'objet sur le « Clic haut ».

```
[ManipulationDirecte, « Objet », « MouseMove », « MouseMove »*, « Clic haut »]
```

Figure 52 : Questionnaire pour la translation d'un objet, où ManipulatioDirecte désigne l'interacteur de Manipulation Directe et où "Objet", "MouseMove", "MouseMove", "Clic haut" désignent la série de jetons attendus*

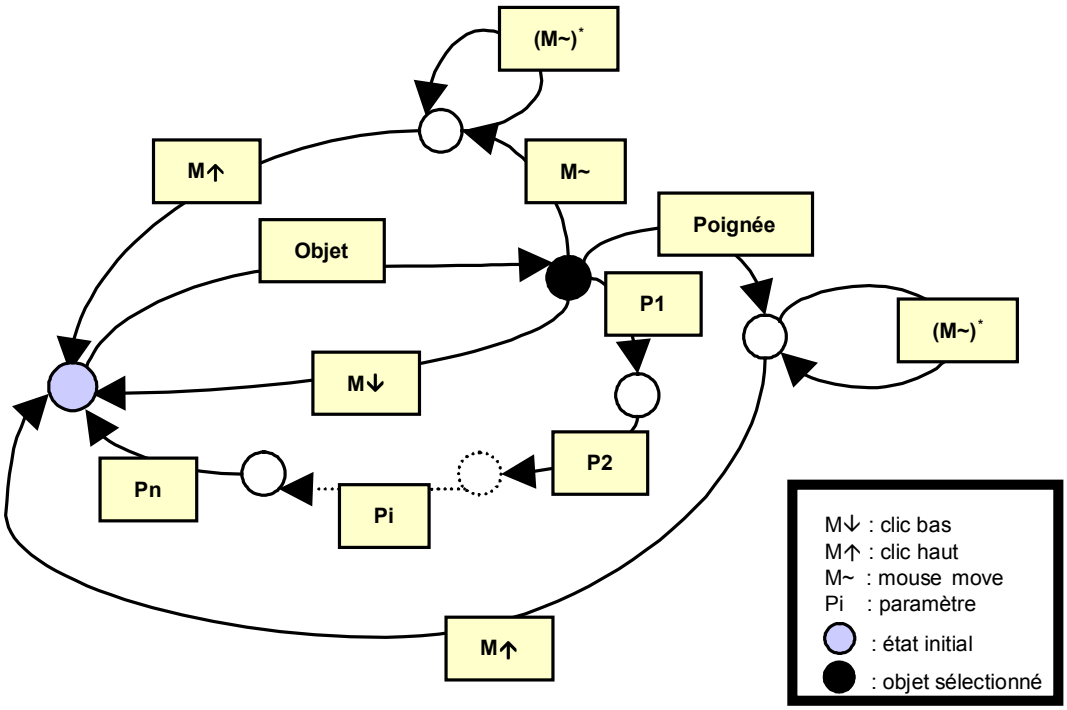


Figure 53 : Automate correspondant à l'interacteur de manipulation directe

Mais, l'interacteur de manipulation directe, tel que nous l'avons défini ici, ne peut pas répondre à tous nos besoins. En effet, lorsqu'il a reçu tous les jetons nécessaires à l'activation d'un questionnaire, un interacteur « vide » son registre (tous les jetons qu'il sauvegarde au fur et à mesure qu'il les reçoit). Dans le cas de la manipulation directe, on voudrait pourtant garder, d'une fois sur l'autre, l'objet sélectionné, car on ne veut pas que l'utilisateur ait à le sélectionner à nouveau après chaque opération de manipulation directe.

Pour cette raison, nous avons modifié le rôle du moniteur en lui adjugeant une nouvelle charge. A la fin d'une interaction par manipulation directe sur un objet donné, l'interacteur de manipulation directe rend un jeton contenant l'objet de la précédente interaction. Le moniteur, a qui ont permet de reconnaître cet interacteur particulier, réinsère immédiatement ce même jeton au même interacteur, plaçant à nouveau celui-ci dans un état susceptible de recevoir des actions de manipulation directe.

2.2.4.3 Placement des interacteurs de manipulation directe

Comme on l'a déjà vu par ailleurs, la place des interacteurs dans la hiérarchie du moniteur revêt une importance capitale : les jetons produits par un interacteur donné ne sont récupérables que par un interacteur placé « au dessus » de lui. De ce fait, il est très important de placer les trois interacteurs de manipulation directe au sommet de la hiérarchie dans cet ordre : l'interacteur de sélection des poignées sous celui de sélection d'un objet, lui même sous celui de manipulation directe proprement dit (Figure 54). Cet ordre est nécessaire pour que :

- **La sélection d'un nouvel objet parvienne toujours à l'interacteur de manipulation directe.** Si l'interacteur de Manipulation Directe est situé sous l'interacteur de sélection d'objet en manipulation directe, l'objet fourni ne sera pas en mesure d'alimenter l'interacteur de manipulation directe.
- **La sélection d'un objet n'empêche pas la sélection d'une poignée.** En règle générale, les poignées d'un objet sont situées sur l'objet lui-même ou au moins dans la zone de sélection de l'objet (un clic très proche de l'objet mais non sur l'objet sélectionne l'objet). De ce fait, tous les clics destinés à sélectionner une poignée peuvent être interprétés comme des clics de sélection de l'objet. Pour éviter cela, il suffit que l'interacteur de sélection de poignée soit placé AVANT celui de sélection d'objet.

D'autre part, l'interacteur de Manipulation Directe doit être situé à l'extrémité supérieure de la hiérarchie (Figure 54) car il consomme chaque objet reçu (rappelons qu'il s'agit d'un interacteur de manipulation directe et que sa « commande » d'activation n'est autre qu'un objet) pour en faire le nouvel objet d'intérêt. Or certains interacteurs de construction utilisent eux aussi des objets en entrée (par exemple, un interacteur chargé de créer un cercle à partir de trois droites). Il ne faut donc pas qu'un objet circulant à travers la hiérarchie soit intercepté par l'interacteur de manipulation directe s'il est destiné à un interacteur de construction.

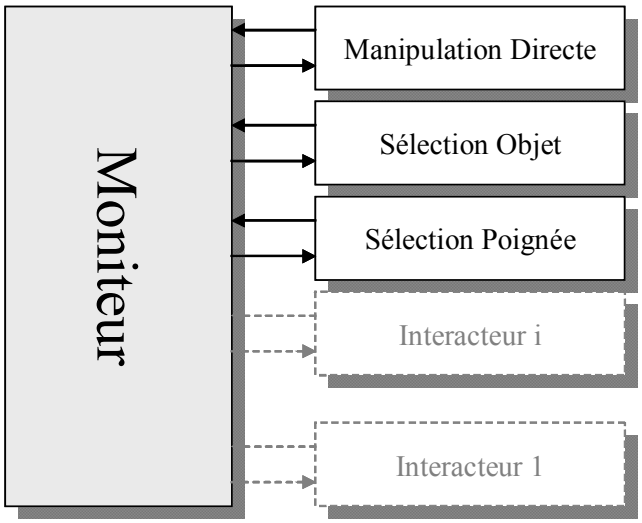


Figure 54 : Organisation des interacteurs dédiés à la manipulation directe à l'intérieur de la hiérarchie du moniteur

2.2.4.4 Mode du dialogue

Pour que la Manipulation Directe n'intervienne pas dans les phases de dialogues structurés, le système doit être en mesure de savoir, à tout moment, dans quel état il se trouve : mode de dialogue structuré ou mode de manipulation directe.

Une première solution consisterait à utiliser une pile d'états. Le système « empilerait » dès qu'un interacteur de production serait activé et « dépilerait » dès qu'un interacteur de

production rendrait un jeton au moniteur. Lorsque la pile serait vide, on considérerait que le système fonctionne en Manipulation Directe, sinon, c'est au contraire le mode de dialogue structuré qui serait actif.

L'inconvénient de cette méthode est qu'elle ne prend pas en compte le fait qu'une action de production peut être abandonnée en cours de route (l'interacteur est alors activé mais ne rend pas de valeur). De ce fait, nous avons adopté une autre approche, qui repose sur le principe suivant :

> ***Mode***. *Pour savoir si l'on se trouve en mode manipulation directe, on « regarde » si un des interacteurs du moniteur attend un « Pointé » graphique (donc un clic haut).*

Malgré sa simplicité, cette définition répond exactement à nos attentes : en supposant que les jetons « clic haut », « clic bas » et « poignée » ne sont utilisés que dans des phases de manipulation directe, la seule ambiguïté qui se pose vient de l'utilisation, au cours d'un dialogue structuré, d'un jeton de type « pointé » (qui peut se confondre avec un « clic haut »). Cette définition permet, du reste, d'utiliser la manipulation directe dans toutes les phases de dialogues complexes, sans les interrompre.

Le rôle du moniteur se trouve augmenté car il est le seul à avoir une connaissance des jetons attendus par les interacteurs à un moment donné de l'interaction. C'est donc lui qui doit dire si le système se trouve dans un mode de manipulation directe ou non.

2.2.4.5 Remarque

Parvenu à ce stade de l'analyse, on constate que l'ajout de ces interacteurs ne résout pas tous les problèmes. En effet, nous manipulons trois types d'événements en provenance de la souris : le clic haut, le clic bas et le déplacement. Lorsque, dans le cadre d'un dialogue structuré, l'utilisateur sélectionne une entité, c'est l'événement clic haut qui est pris en compte pour fournir la position à partir de laquelle on calculera l'objet le plus proche du curseur de la souris. Or, si l'on s'en tient à ce que l'on a fait, le clic bas précédant immanquablement le clic haut traversera la hiérarchie pour débuter éventuellement une action de manipulation directe. Le clic haut suivant, lui, n'arrivera jamais jusqu'à l'interacteur de manipulation directe, puisqu'il sera consommé pour la sélection d'un objet dans le cadre du dialogue structuré. De ce fait, le dialogue du système sombrera dans un état incertain.

Pour résoudre ce problème nous proposons d'ajouter un filtre à notre contrôleur de dialogue.

2.2.4.6 Filtre

2.2.4.6.1 Description

Le filtre est un composant logiciel, placé entre le moniteur et l'adaptateur de présentation (Figure 55), chargé d'analyser les entrées fournies par la couche de présentation et de les

trier en fonction de l'état du dialogue dans lequel se trouve le système (mode manipulation directe ou non manipulation directe).

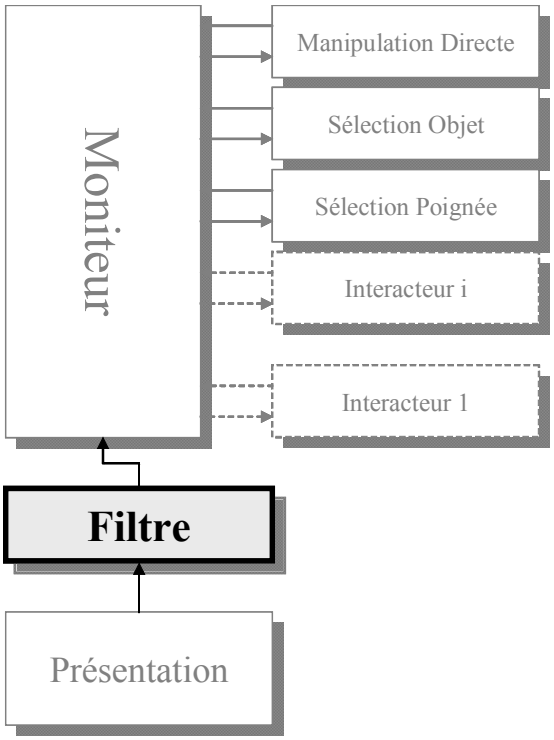


Figure 55 : Le filtre de Manipulation Directe

Pour savoir si l'on se trouve en mode manipulation directe, on « regarde » si un des interacteurs du moniteur attend un « Pointé » graphique (donc un clic haut). Si c'est le cas, on suspend le mode « Manipulation Directe » et on ne transmet au moniteur ni les clic bas, ni les déplacements de souris. Dans le cas contraire, on laisse passer tous les types de jetons (Figure 56).

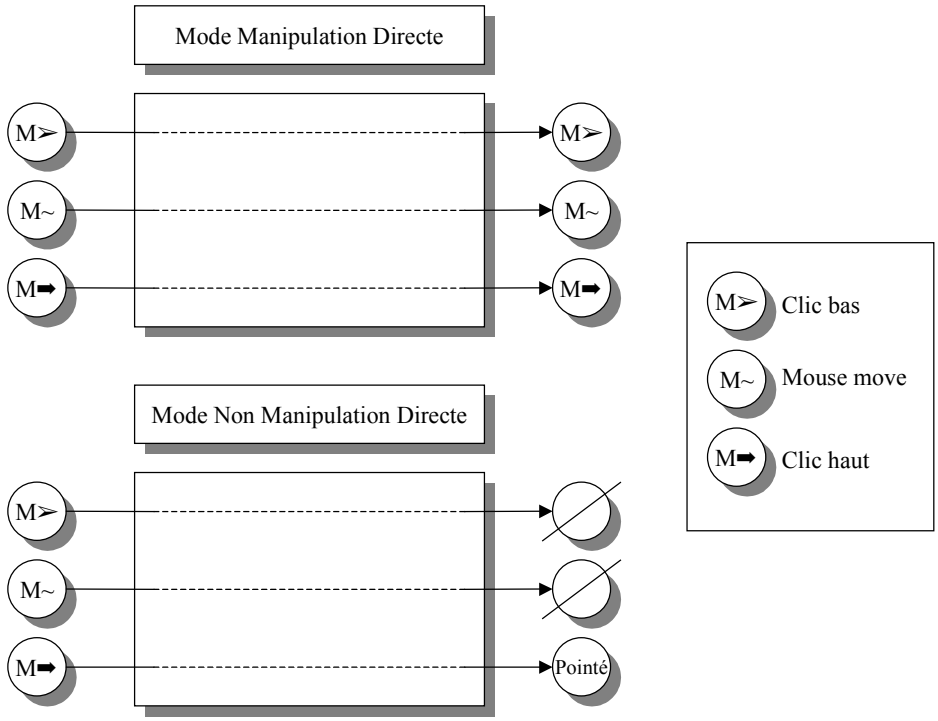


Figure 56 : Fonctionnement du filtre

2.2.4.6.2 Fonctionnement

Le filtre reçoit des entrées de la couche de présentation. Selon le mode dans lequel se trouve le système, il laisse ou non passer les jetons de manipulation directe (clics bas et déplacement de souris). Le moniteur fait circuler les jetons à travers la hiérarchie des interacteurs au sommet de laquelle se trouvent les trois interacteurs de manipulation directe. Parmi eux, deux interacteurs de sélection sont chargés de transformer les clics bas qu'ils reçoivent en objets ou en poignées consommables uniquement par l'interacteur de manipulation directe. Cet interacteur est distingué parmi les autres par le moniteur. Celui-ci est ainsi en mesure, à chaque fin d'interaction directe sur un objet, de ré-insuffler l'objet en question afin qu'il reste l'objet « en cours » jusqu'à ce qu'un nouvel objet prenne sa place.

2.2.4.7 Récapitulatif

Rôles du moniteur

- Il connaît l'interacteur de Manipulation Directe et analyse ses entrées ;
- Il assure que les interacteurs de sélection de MD restent « actifs » ;
- Il met à jour le mode (MD ou non) en espionnant les jetons qu'il transmet et leur destinataire.

Fonctionnement

- Le filtre reçoit des jetons de l'Adaptateur de Présentation ;
- Selon le mode, il les transmet ou non au Moniteur ;
- Le moniteur active les interacteurs de sélection pour la Manipulation Directe ;

- Dès qu'un interacteur de production est activé ou rend un jeton, le moniteur met à jour le mode (Manipulation Directe ou non) ;
- Le moniteur active et réinitialise l'interacteur de Manipulation Directe.

2.2.4.8 Construction

L'implémentation de la Manipulation Directe dans H4 nous a obligé à modifier légèrement le Moniteur. Celui-ci possède désormais une méthode permettant de renseigner le système sur le mode de dialogue dans lequel il se trouve. D'autre part, nous avons du autoriser le Moniteur à différencier l'interacteur de Manipulation Directe des interacteurs « classiques ». Alors qu'il n'y avait jusqu'à maintenant aucune différence entre les interacteurs, cette propriété assurant l'indépendance des interacteurs les uns par rapport aux autres, le Moniteur « sait » désormais que le dernier, dans la hiérarchie, représente un interacteur particulier. Et celui-ci est ainsi traité de manière particulière puisque le Moniteur lui ré-insuffle systématiquement l'objet d'intérêt lorsqu'il revient dans son état initial (ceci afin de ne pas avoir à sélectionner l'objet d'intérêt après chaque nouvelle manipulation).

Ceci étant, les modifications apportées au Moniteur n'influent en rien sur le fonctionnement « classique » du contrôleur de dialogue. Les interacteurs habituels conservent leur indépendance et leur anonymat par rapport au Moniteur.

Pour spécifier les comportements de manipulation directe, le concepteur de l'application doit donc, à partir des interacteurs prédéfinis que sont les trois décrits précédemment :

- Personnaliser la méthode de sélection d'un objet, appelée lors de l'activation du questionnaire. A partir de la position fournie, la méthode doit être capable de trouver l'objet le plus proche, de créer les poignées correspondantes et de les afficher.
- Personnaliser la méthode de sélection d'une poignée.
- Ajouter tous les questionnaires nécessaires au fonctionnement de l'interacteur de Manipulation Directe. C'est ici que le concepteur spécifie les différents comportements en fonction des jetons reçus, à partir d'un objet sélectionné.

2.2.5 Comportements supplémentaires

En dehors des comportements que nous nous étions fixé au départ, la modélisation utilisée pour la Manipulation Directe dans H4 nous permet d'envisager facilement l'intégration d'autres procédés.

2.2.5.1 Modification des attributs d'un objet

Modifier les attributs d'un objet consiste à accéder à un certain nombre de ses propriétés par l'intermédiaire d'un panneau de configuration. L'objet est tout d'abord sélectionné de la manière habituelle, puis, en double cliquant sur celui-ci ou en effectuant un clic droit, on accède à un panneau permettant de modifier la valeur de ses attributs.

Ce comportement peut très facilement être obtenu par la modélisation que nous avons choisie. Il suffit d'ajouter aux jetons déjà présentés, un jeton représentant le double clic ou le clic droit. Ensuite, à l'intérieur de l'interacteur de Manipulation Directe, le concepteur de l'application doit simplement ajouter un questionnaire permettant d'activer le panneau relatif à l'objet sélectionné lorsque ce type de jeton est présenté à l'interacteur.

2.2.5.2 Sélection multiple

Nous n'avons étudié ici que la sélection d'un objet unique. Pour permettre la sélection de plusieurs objets, il suffit d'utiliser un nouveau type de jeton, capable de transporter une liste d'objets. La sélection par manipulation directe serait modifiée pour fournir non pas un objet unique mais un jeton « liste d'objets ». Il faut modifier de la même façon l'interacteur de manipulation directe pour que l'objet d'intérêt devienne une liste d'objets d'intérêt.

2.2.6 Conclusion

L'architecture H⁴ est dédiée au dialogue structuré. Elle a été mise au point afin de permettre au concepteur d'une AGICT, comme un système de CAO par exemple, de modéliser facilement la décomposition des tâches en un ensemble de buts/Sous-buts. Ceci étant, toutes les applications interactives, aussi techniques soient-elles, n'ont pas systématiquement besoin de ce type de dialogue pour exprimer les besoins de l'utilisateur. Ainsi, il paraît extrêmement regrettable que H⁴ ne soit pas en mesure de procurer des modes d'interactions plus simples, comme la Manipulation Directe.

Malgré l'antagonisme apparent entre le dialogue structuré (préfixé) et la Manipulation Directe (post-fixée), nous avons montré comment il était possible, en modifiant le rôle du Moniteur, et en ajoutant des interacteurs spécialisés et parfaitement identifiés, de faire cohabiter ces deux modes de dialogue.

Notre approche répond aux critères établis en préambule de ce travail. D'abord, le passage d'un mode à l'autre est transparent. D'autre part, la Manipulation Directe n'amène pas de taux d'erreur supplémentaire puisque aucune confusion n'est permise entre l'appel des mêmes opérations dans un mode et dans l'autre. Enfin, l'intégration n'amène pas de travail particulièrement pénible au développeur : les jetons sont prédéfinis, les trois interacteurs supplémentaires sont déjà spécifiés. Il ne reste au concepteur qu'à écrire les méthodes appelées pour la sélection des objets et des poignées et à définir les questionnaires à ajouter à l'interacteur de Manipulation Directe.

Guillaume Patry ([Patry 1999]) reconnaît lui aussi la présence nécessaire de nouveaux interacteurs. Mais sa solution reste floue sur un certain nombre de points essentiels dans le cadre de l'implémentation de H⁴ que nous avons utilisée. Il ne précise par exemple pas quel est le composant responsable du passage dans un mode de Manipulation Directe et ne s'intéresse pas à l'activation des interacteurs par des jetons autres que les « jetons commandes ».

Notre solution détaille elle le rôle du Moniteur de manière claire et formalise le fonctionnement des trois interacteurs identifiés. D'autre part, nous décrivons le fonctionnement d'un filtre nécessaire au bon fonctionnement de notre approche.

La partie suivante présente un outil interactif permettant de générer de manière simple une partie du code du contrôleur de dialogue d'une application basée sur H4.

2.3 DTS Edit : un éditeur pour la Boîte à Outil du Dialogue

2.3.1 Introduction

Tous les développements que nous avons eu à réaliser dans le cadre de cette thèse ont eu pour support l'architecture H4, et plus précisément une implémentation particulière du contrôleur de dialogue de H4 réalisée par Guillaume Texier [Texier, Depaulis, & Guittet 2001 ; Texier & Guittet 1999a ; Texier & Guittet 1999b ; Texier, Guittet, & Girard 2001]. Cette implémentation définit une véritable Boîte à Outils du dialogue, grâce à laquelle un concepteur peut réaliser le dialogue d'une application exactement de la même manière qu'il construirait une interface avec l'aide d'une Boîte à Outils « classique ». Les éléments de cette Boîte à Outils ne sont pas des réifications d'éléments de présentation, mais des réifications d'éléments du dialogue.

La Boîte à Outils du dialogue implémente non seulement tous les éléments du contrôleur de dialogue de H4 décrit dans le chapitre 2 (jetons, interacteurs, questionnaires et moniteur) sous forme de classes, mais elle inclut en plus un générateur automatique permettant au système de créer lui-même les automates représentant les interacteurs à partir de la définition des questionnaires.

Par exemple pour créer un interacteur permettant à un utilisateur de créer des cercles, il faut d'abord créer le questionnaire correspondant à la suite de jetons d'interaction attendu pour activer la méthode correspondante. Ensuite, il faut ajouter ce questionnaire à un interacteur, puis intégrer celui-ci à la hiérarchie du Moniteur. L'exemple suivant illustre la syntaxe qui doit être utilisé pour créer un interacteur composé de deux questionnaires. Le premier permet la création d'un cercle par deux points, et l'autre celle d'un cercle par un point et une valeur (correspondant chacun au centre et au rayon). Cet exemple ne fait pas apparaître les fonctions du noyau fonctionnel associées au questionnaire et qui construisent de manière effective les cercles.

```
Questionnaire Q_Cercle_P_N (« create_circle », {« position », « numeric »})
// "circle creation by center & radius" Questionnaire specification
Questionnaire Q_Cercle_P_P (« create_circle », {« position », « position »})
// " circle creation by two points" Questionnaire specification
Diaget Creation (« Creation », {Q_Circle_P_N, Q_Circle_P_P})
// The Questionnaires are added to the « creation » Diaget. The
// Transition Net. is generated from the questionnaire specifications.
Monitor my_monitor ({Creation, …})
// The Diagets are added to the monitor hierarchy.
// The adding order sets the hierarchy.
```

2.3.2 *Problèmes*

La Boîte à Outils du Dialogue permet au concepteur d'une application de créer relativement facilement le contrôleur de dialogue de son système, dans un mode bien connu des programmeurs d'interface. Ceci étant, comme cela est déjà le cas avec les Boîtes à Outils graphiques, la programmation en est fastidieuse et répétitive. Comme on peut le constater dans l'exemple du paragraphe précédent, le code est répétitif et l'utilisation du « copier/coller » est une source fréquente d'erreur. D'autre part, la structure même du contrôleur de dialogue de H⁴ possède un certain nombre de propriétés dont l'analyse apporte des informations sur la complétude des tâches. Ces propriétés sont difficilement exploitables « à la main » mais peuvent très bien être vérifiées de manière automatique.

- **Consistance de production et de consommation.** Un questionnaire appartenant à un interacteur donné est-il en mesure de recevoir un certain type de jeton ? Pour cela, il faut qu'au moins un des interacteurs le précédant dans la hiérarchie du Moniteur soit capable de produire un tel type de jeton. Si ce n'est pas le cas, la tâche du système correspondant à ce questionnaire ne sera jamais activée.
- **Questionnaires et jetons de commande.** Est-ce que deux questionnaires appartenant à deux interacteurs différents possèdent la même signature ? Une telle situation empêche celui placé dans l'interacteur le plus élevé dans la hiérarchie du Moniteur, et ainsi la tâche du système correspondant de s'exécuter.
- **Surcharge.** Est-ce que deux questionnaires du même interacteur possède le même début de séquence de jetons ? Dans ce cas, une des deux tâches ne pourra jamais s'exécuter.

Toutes ces propriétés peuvent être facilement vérifiées grâce par un outil.

Pour cette raison, ainsi que pour faciliter le long et pénible travail de codage, nous avons mis au point un outil graphique et interactif permettant au concepteur d'une application de générer une partie du contrôleur de dialogue de H⁴. et de vérifier plusieurs propriétés de complétude sur les tâches modélisées.

2.3.3 *DTS Edit (Dialog ToolSet Editor)*

DTS Edit ([Depaulis, Maiano, & Texier 2002]) (pour **D**ialog **T**ool**S**et **Edit**or ou Editeur de Boîte à Outils du Dialogue) est un environnement de développement graphique et interactif dont la fenêtre principale se décompose en deux parties (Figure 57).

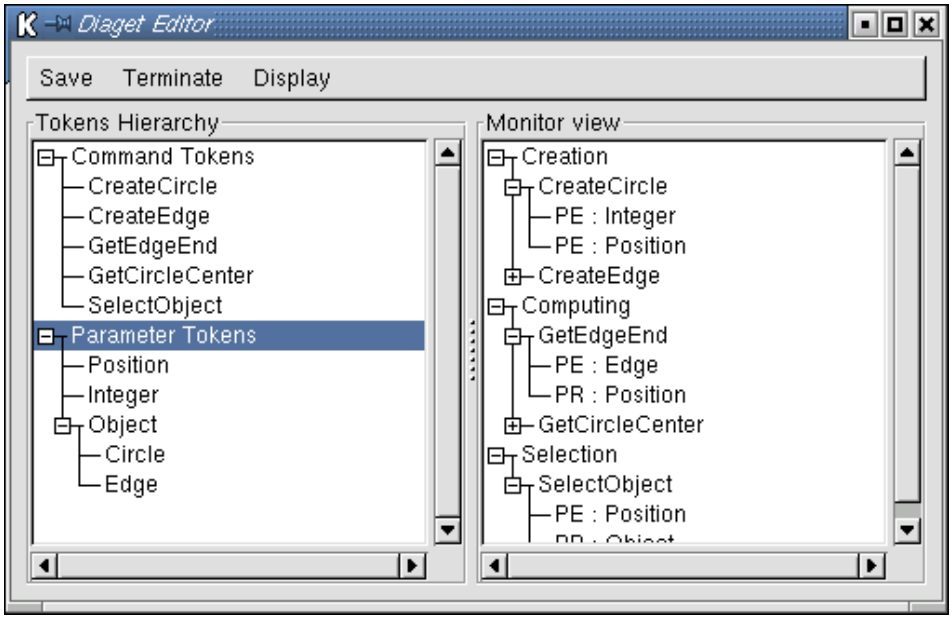


Figure 57 : Fenêtre principale de DTS Edit

La partie gauche affiche toutes les informations sur les jetons. Elle permet notamment de séparer les jetons « commandes », des jetons paramètres. Ceux-ci sont présentés sous forme d'arbre, correspondant à la hiérarchie choisie par le concepteur.

La partie droite est quant à elle chargée de représenter les interacteurs et la signature des questionnaires qui les composent. L'ordre de présentation (haut vers bas) à une importance sémantique puisque qu'elle correspond à la hiérarchie du Moniteur.

Cet éditeur utilise largement la manipulation directe et autorise notamment des opérations telles que le « drag'n drop » d'un panel à l'autre. L'utilisateur a également la possibilité de sauvegarder et de charger une architecture, que ce soit dans le format de l'éditeur ou dans celui du code généré.

Voyons maintenant comment un concepteur peut, à partir de cet éditeur, réaliser toutes les opérations (création de jetons, d'interacteurs, de questionnaires, etc.) permettant la description du contrôleur de dialogue de son application et effectuer les différentes vérifications présentées précédemment.

2.3.3.1 Jetons

La partie gauche de l'éditeur présente la hiérarchie des jetons de l'application. Cette hiérarchie est séparée en deux classifications distinctes.

D'un côté, on trouve les jetons « commande ». Ils transportent le nom des questionnaires et sont, dans l'implémentation initiale du Contrôleur de Dialogue de H[4], les seuls jetons capables d'initier l'automate d'un interacteur : à partir du moment où l'interacteur reçoit une commande correspondant à un de ses questionnaires, celui-ci est alors en mesure de recevoir les autres jetons de sa séquence d'entrée. Dans la plupart des applications, ils sont fournis par l'utilisateur au Contrôleur de Dialogue par l'intermédiaire des menus ou des boutons de l'interface graphique.

Dans une nouvelle hiérarchie, l'interface de l'éditeur présente les jetons « paramètre ». Ils s'agit de jetons transportant les valeurs des objets du noyau fonctionnel (par exemple des cercles et des segment dans une application de type MacDraw) et de la couche de présentation (par exemple des pointés graphiques). Ces jetons sont organisés de manière hiérarchique, exactement comme le sont les objets du Noyau Fonctionnel de l'application. Par exemple, dans une application de type MacDraw, l'utilisateur peut créer des cercles et des segments dont on peut imaginer qu'ils héritent chacun d'un objet de plus haut niveau d'abstraction, « Objet Graphique ». L'organisation des objets du Noyau Fonctionnel doit être transmise aux jetons afin que le dialogue puisse profiter des mêmes avantages que le modèle. Ainsi, des questionnaires peuvent être regroupés entre eux. Dans l'exemple de l'application de type MacDraw, il n'est pas nécessaire de définir plusieurs questionnaires de sélection : un seul peut regrouper la sélection de tous les objets graphiques, qu'ils soient des cercles ou des segments. Grâce à la relation d'héritage, le questionnaire peut se contenter de porter sur un jeton correspondant aux « Objets Graphiques » et être utilisé aussi bien pour les cercles que pour les segments.

Dans DTS Edit, les deux classifications de jetons sont affichées à l'écran. Les commandes sont rangées dans un arbre à un seul niveau de profondeur (il n'y a pas d'héritage entre les jetons « commande »). Ces commandes sont créées de manière automatique. En effet, chaque commande correspond à l'activation d'un questionnaire particulier. De ce fait, à chaque nouvelle création de l'un d'eux, il suffit de créer le jeton « commande » associé, en lui donnant exactement le nom du questionnaire.

Les jetons « paramètre » sont pour leur part représentés sous la forme d'un arbre représentant les relations d'héritage. Chaque élément d'un sous-arbre hérite des propriétés de l'élément racine dont le sous-arbre est issu. Contrairement aux jetons « commande », les paramètres ne sont pas créés automatiquement. Pour définir un nouveau jeton de ce type, l'utilisateur doit définir son nom et spécifier s'il existe une relation d'héritage avec un type de jeton existant. Le nouveau jeton est alors ajouté dans la hiérarchie (Figure 58). Il est important de noter que les jetons paramètres doivent d'abord être définis dans cette partie pour pouvoir être utilisés lors de la spécification des questionnaires.

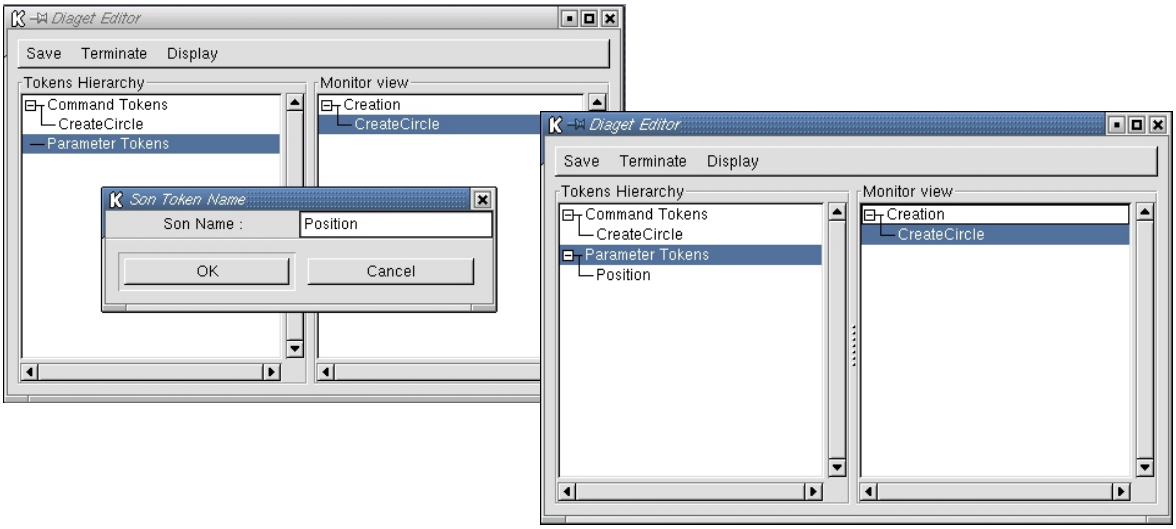


Figure 58 : Création d'un jeton "paramètre"

2.3.3.2 Moniteur

La partie droite de la fenêtre principale représente le Moniteur du Contrôleur de Dialogue. Sur ce panel, le concepteur peut créer les interacteurs et les questionnaires qui les composent, ainsi que modifier l'organisation générale du Moniteur.

Pour créer un interacteur, l'utilisateur donne simplement une chaîne de caractères correspondant à sa désignation (Figure 59). Les interacteurs sont affichés selon la hiérarchie du Moniteur : les plus hauts dans la hiérarchie apparaissent au somment du panel.

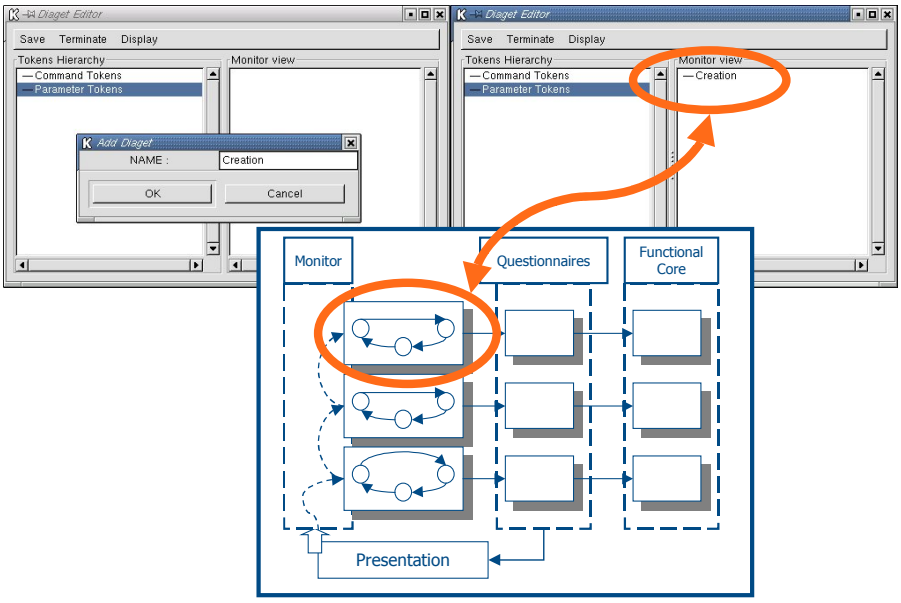


Figure 59 : Creation d'un interacteur et conséquences[1] sur le contrôleur de dialogue

Le concepteur de l'application peut alors ajouter un questionnaire à l'interacteur ainsi défini. Pour ça, il donne encore simplement une chaîne de caractères correspondant à sa désignation. Cette action a pour effet de générer automatiquement un jeton « commande », qui apparaît simultanément dans la liste des commandes de la partie gauche de la fenêtre (Figure 60).

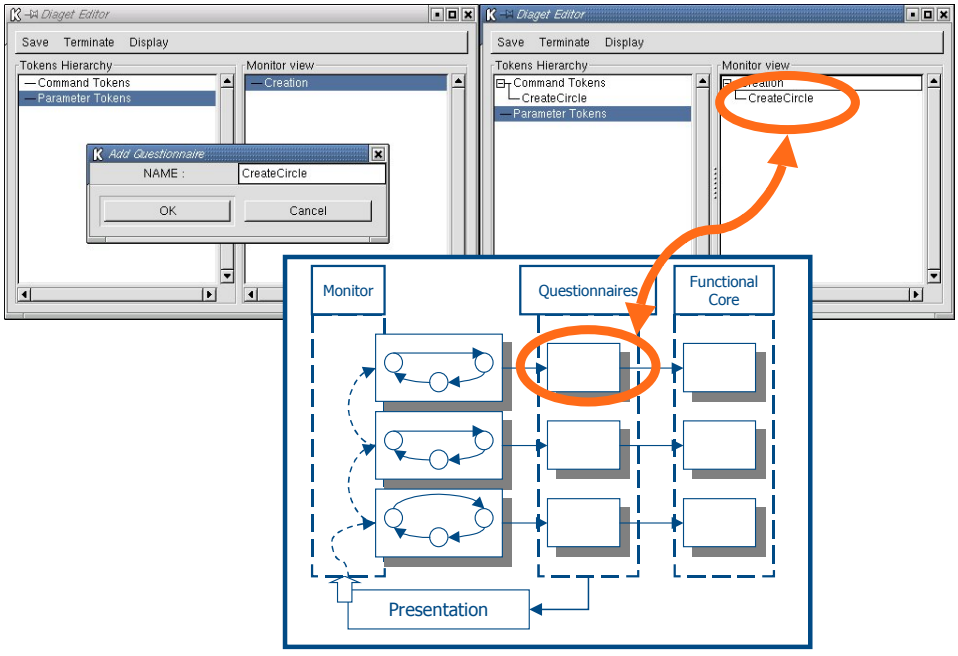


Figure 60 : Ajout d'un questionnaire à un interacteur et conséquences au niveau du contrôleur de dialogue

[1] Les conséquences sur le Contrôleur de Dialogue, illustrées par les flèches sur cette figure et sur les suivantes, ne sont effectives que lorsque le concepteur décide explicitement de générer le code correspondant.

L'étape suivante de la spécification du Contrôleur de Dialogue consiste à spécifier la signature du questionnaire. Un menu permet au concepteur d'ajouter des paramètres d'entrée et éventuellement un paramètre de sortie aux questionnaires existants. Les types de jetons utilisables pour cette action sont évidemment ceux qui ont été créés et qui apparaissent dans la partie gauche de la fenêtre (Figure 61).

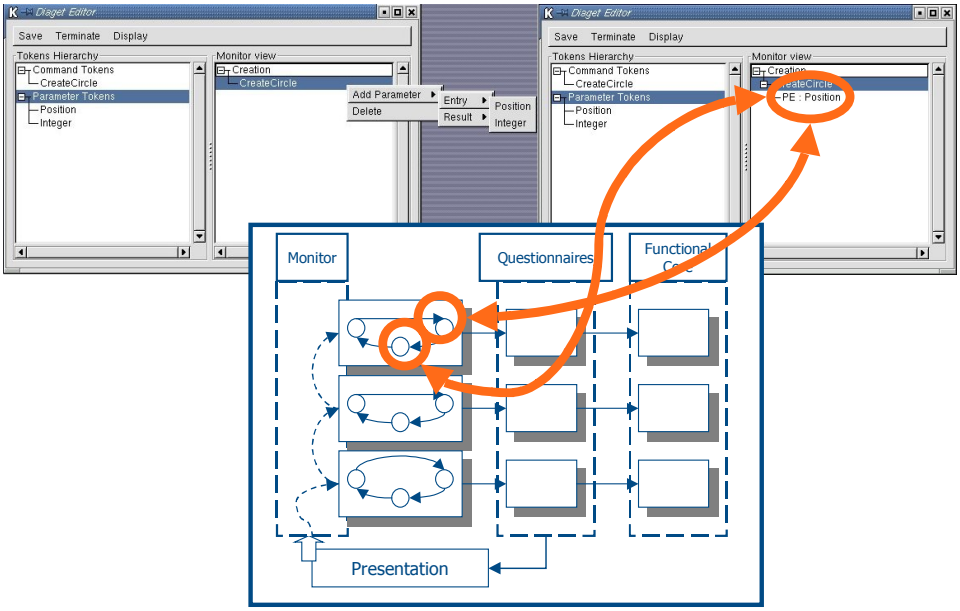


Figure 61 : Spécification de la signature d'un questionnaire et conséquences sur le contrôleur de dialogue

DTS Edit n'est évidemment pas capable de générer seul le code des questionnaires. La spécification des signatures permet au logiciel de créer les automates représentant les différents interacteurs, en fonction de ces questionnaires. Les questionnaires eux-mêmes, qui font la liaison entre le Contrôleur de Dialogue et le Noyau Fonctionnel, doivent être programmés « à la main » par le concepteur. La seule aide qu'apporte DTS Edit à ce niveau de conception est un éditeur de texte permettant de composer les fichiers des questionnaires.

La dernière aide interactive qu'apporte l'éditeur est la possibilité de modifier interactivement, par des opérations de « drag'n drop », la hiérarchie des interacteurs (Figure 62).

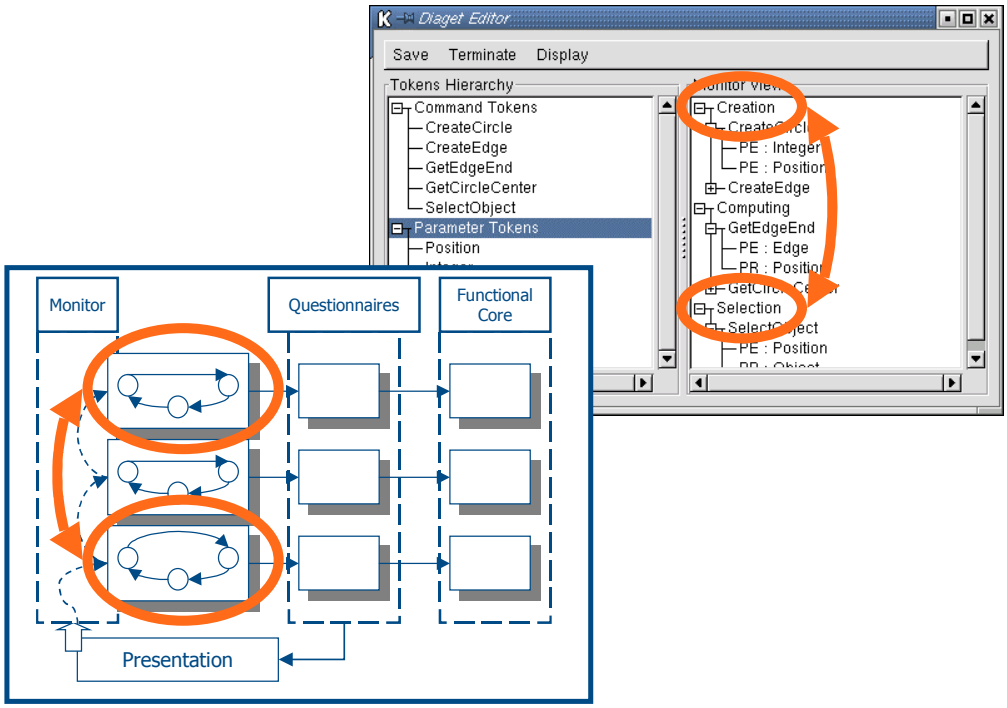


Figure 62 : Modification de la hiérarchie des interacteurs par manipulation directe et conséquences sur le contrôleur de dialogue

Une fois que le concepteur a terminé la description de tous ses jetons, qu'il a spécifié les interacteurs et la signature des différents questionnaires qui le composent, il peut alors générer le code du Contrôleur de Dialogue de son application en appuyant simplement sur un bouton. C'est à cette étape du processus que l'éditeur opère toutes les vérifications sur la complétude des tâches décrites.

2.3.3.3 Vérifications des propriétés de complétude des tâches

Grâce à l'organisation du contrôleur de dialogue de H4, il est relativement aisé de vérifier plusieurs propriétés du dialogue.

2.3.3.3.1 Consistance de production et consommation

Dans H4, les interacteurs sont organisés verticalement : ceux placés au sommet de la hiérarchie sont en mesure de recevoir des jetons produits par des interacteurs de plus bas niveau. Dans cette organisation, l'interacteur placé tout en bas ne peut recevoir des jetons que de la couche de présentation, et inversement, celui placé au sommet voit ses éventuelles productions systématiquement perdues. En se basant sur le classement des interacteurs, sur les jetons qu'ils produisent et sur ceux qu'ils consomment, DTS Edit est capable de prévenir l'utilisateur dès qu'un problème est détecté :

- Si un des questionnaires de l'interacteur de plus haut niveau émet un jeton en sortie ; l'utilisateur ne peut ainsi pas placer par erreur un interacteur de production au sommet de la hiérarchie ;

- Soit un questionnaire Q appartenant à un interacteur I et attendant un jeton de type T. Si, dans la hiérarchie des interacteurs, il n'existe aucun interacteur placé SOUS I et capable de produire un jeton de type T, alors Q ne sera jamais complété et la tâche associée jamais exécutée. L'utilisateur est donc prévenu de cette éventuelle erreur et ne peut pas par, erreur, provoquer la « famine » d'un questionnaire.

2.3.3.3.2 Questionnaires et jetons « commande »

Un questionnaire est en mesure de recevoir la série de jetons qu'il le définit uniquement lorsqu'il a préalablement été activé par une commande. Celle-ci est habituellement fournie par la couche de présentation, grâce à un menu ou un bouton de l'interface. Si deux questionnaires, placés dans deux interacteurs différents, ont le même nom, l'un d'entre eux ne pourra jamais être activé. En effet, le jeton « commande » sera soumis à tous les interacteurs dans l'ordre de la hiérarchie, en commençant celui placé le plus bas, mais elle sera toujours interceptée par le même. Celui des deux interacteurs placés le plus bas dans la hiérarchie se verra toujours proposer le jeton en premier, et sera toujours en mesure de l'accepter. DTS Edit prévient l'utilisateur lorsqu'un cas semblable se produit.

2.3.3.3.3 Surcharge

Dans le même interacteur, deux questionnaires peuvent porter le même nom. Il s'agit d'une surcharge autorisée qui permet à l'utilisateur du système de commander la création d'un cercle par un unique bouton, qu'il définisse ensuite ce cercle par deux positions ou par une position et une valeur numérique. Cependant, un problème peut survenir si les paramètres d'entrée d'un des questionnaires et une sous-liste des paramètres d'entrée de l'autre. Dans ce cas, il est en effet impossible au contrôleur de dialogue de savoir quelle attitude adopter :

- Si lorsqu'il a reçu la liste de jetons du premier questionnaire, il décide de déclencher l'action correspondante (ce qui est possible puisque tous les paramètres sont là), cela signifie que, quoiqu'il arrive, le second ne se verra jamais proposer aucune séquence de jetons et il sera par conséquent virtuellement « gelé » ;
- Si par contre, il attend l'arrivée de nouveaux jetons, essayant ainsi de compléter la liste du second questionnaire, le premier se trouve alors « oublié » et l'action correspondante ne sera jamais exécutée.

2.3.4 Conclusion

La création du contrôleur de dialogue d'une application modélisée selon l'architecture H[4] est un travail lourd et fastidieux de programmation. Le caractère répétitif des instructions mène souvent le concepteur à des erreurs dues par exemple aux nombreux copier/coller utilisés. D'autre part, l'organisation des interacteurs entre eux propose des propriétés permettant théoriquement de vérifier la validité et la complétude des tâches modélisées. Malheureusement, un tel travail n'est pas envisageable « manuellement ».

Pour ces raisons, nous avons mis au point un outil interactif qui répond aux besoins du concepteur : il génère une grande partie du code du contrôleur de dialogue en permettant une définition aisée des différents composants. D'autre part, il est capable de vérifier différentes propriétés de complétude des tâches en s'appuyant sur les propriétés et l'organisation des interacteurs.

2.4 Conclusion

L'architecture H4 répond aux besoins d'une application de conception technique en proposant un mode de dialogue structuré. La structure du contrôleur de dialogue permet par exemple de rendre les tâches modélisées indépendantes entre elles et de les faire communiquer facilement. L'organisation utilisée autorise ainsi des combinaisons « inter-tâches » qui n'ont pas besoin d'être prévues par le concepteur de l'application, ce qui entraîne une richesse d'expression unique au niveau du dialogue entre le système et l'utilisateur.

Malheureusement, ce dialogue ne se prête a priori pas à des formes d'échange plus simple comme la manipulation directe, pourtant indispensables dans tout système interactif digne de ce nom. D'autre part, la programmation du contrôleur de dialogue, est répétitive, longue et fastidieuse, provoquant de nombreuses erreurs, tant au niveau syntaxique, qu'au niveau de la composition et de l'organisation des interacteurs.

En modifiant le rôle du Moniteur et en ajoutant des interacteurs dédiés, nous avons montré dans cette partie comment il était possible de faire cohabiter les modes de dialogues structurés et par manipulation directe au sein du contrôleur de dialogue de H4. Nous avons également expliqué comment un outil graphique et interactif avait été mis en place pour permettre au concepteur d'une application reposant sur H4 de définir facilement les différents composants du contrôleur de dialogue, ainsi que d'en vérifier certaines propriétés.

Ces introspections nous ont été utiles pour permettre d'outiller le moteur générique de simulation de métamorphoses que nous avons développé et qui fait l'objet de la dernière partie de ce mémoire.

3 Un moteur générique de simulation de métamorphoses

3.1 Introduction

Comme nous l'avons vu précédemment, la création de métamorphoses d'objets naturels structurés passe en règle générale par un contrôle algorithmique. Pour décrire un tel contrôle, les systèmes existants ne proposent d'autres solutions que celle consistant à programmer textuellement l'enchaînement des différentes instructions qui le composent.

Pour résoudre ce problème, nous avons choisi d'utiliser la programmation sur exemple. Parmi les différentes solutions qui s'offraient à nous, la plus évidente consistait bien en-

tendu à créer un système entier « prêt à l'utilisation ». Celui-ci aurait reposé sur un modèle donné permettant de représenter un certain nombre d'objets naturels et de métamorphoses correspondantes. Nous aurions construit un système de programmation sur exemple autour de celui-ci.

Mais nous avons voulu aller plus loin. Pour cette raison, nous avons abstrait les propriétés communes (dans le contexte de la création de métamorphoses) des méthodes basées sur la topologie afin de mettre au point un modèle générique sur lequel pourrait se greffer tout modèle de représentation d'objet naturel structuré. En nous basant sur ce modèle générique, nous avons pu développer un moteur d'enregistrement interactif et de rejeu s'appuyant sur certaines de ces caractéristiques particulières. Le résultat est un environnement de développement qui, bien qu'encore peu outillé, permet au concepteur d'une application de simulation de métamorphoses d'objets naturels structurés d'intégrer aisément des concepts de programmation sur exemple. Ceux-ci autorisent ensuite un utilisateur du système ainsi développé à enregistrer et à rejouer interactivement des programmes correspondant à des simulations de métamorphoses.

Cette dernière partie commence par une rapide synthèse des méthodes d'enregistrement et de rejeu de différents systèmes de programmation sur exemple. Nous expliquons en quoi le contexte de notre étude nous incite à utiliser une approche originale. Nous décrivons ensuite successivement le moteur de programmation sur exemple et le modèle générique sur lequel il repose. Enfin, nous présentons une application développée à partir d'un modèle de noyau fonctionnel permettant de représenter des croissances d'arbre en 1D½, et qui utilise notre cadre de développement.

3.2 Originalité de l'approche

L'approche que nous nous proposons d'utiliser présente la particularité, par rapport aux systèmes de programmation sur exemple existants, de ne pas être liée à un modèle particulier. Le moteur de programmation sur exemple raisonne en effet sur les propriétés d'un modèle générique sur lequel doit se greffer le concepteur de l'application.

Nous nous sommes livrés à une étude de plusieurs systèmes de PsE afin de comprendre les liens qu'ils tissent entre le moteur d'enregistrement et le modèle du Noyau Fonctionnel. Nous avons également analysés leurs principales caractéristiques au niveau des méthodes interactives utilisées pour l'enregistrement, de l'intégration des structures de contrôle et de la généralisation des programmes. Cette étude porte exclusivement sur des systèmes développés au LISI, car il s'agit des seuls dont le fonctionnement nous est parfaitement connu, les concepts qui nous intéressent n'étant jamais abordés dans la description des autres systèmes.

3.2.1 LIKE

Description

LIKE est un système de Programmation sur Exemple créé en 1992 par Patrick Girard. Il s'agit d'un Enregistreur de Macro évolué greffé sur un système de CAO.

Enregistrement des programmes

Les programmes sont enregistrés de manière explicite : l'utilisateur commande explicitement le démarrage et l'arrêt de l'enregistrement du programme (exactement comme pour une macro). LIKE enregistre les actions de l'utilisateur au niveau de l'Adaptateur de Présentation (Figure 63). Ceci pose un problème dans le cadre de la ré-exécution puisque toute modification du dialogue invalide les programmes construits précédemment. Par contre, étant donné que l'enregistreur ne connaît rien de la sémantique des actions enregistrées, on peut envisager de le transférer sur une autre application.

Généralisation de l'exemple

Pour généraliser l'exemple, LIKE s'appuie sur la notion de contexte dynamique : chaque sous programme possède un contexte, utilisable par un sous programme appelant uniquement globalement.

Structures de contrôle

Dans LIKE, le déroulement du programme est implicite. Le système est capable de gérer des conditions et des boucles, au niveau de l'Adaptateur de Présentation.

Il n'y a pas de vérifications sémantiques puisque LIKE ne connaît pas le Noyau Fonctionnel.

Liens entre le système de PsE et le Noyau Fonctionnel

Le système d'enregistrement ne connaît rien du Noyau Fonctionnel. Il enregistre uniquement une suite de commande de la couche de présentation. (au mieux, LIKE connaît les types de base de l'application).

Généralisations possibles au niveau de ce qui est lié au NF

Encore une fois, rien, dans ce qui constitue LIKE, n'est lié d'une manière ou d'une autre au Noyau Fonctionnel. C'est tout l'avantage de LIKE par rapport à des systèmes plus évolués comme EBP ou TexAO.

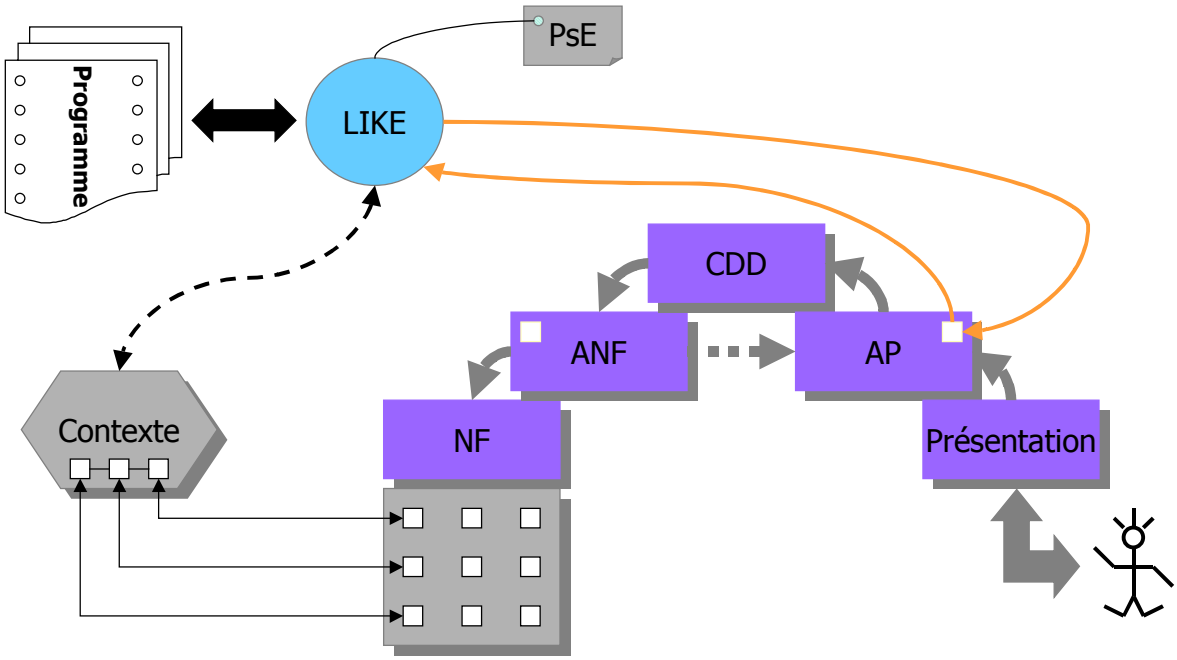


Figure 63 : Architecture du système LIKE

3.2.2 EBP

Description

EBP est un système de Programmation sur Exemple créé en 1993 par Jean-Claude Potier. C'est un mini-système de CAO qui permet à un utilisateur de créer des familles de composants. Le but principal recherché est la génération de programmes neutres.

Enregistrement des programmes

Les programmes sont enregistrés de manière explicite : l'utilisateur commande explicitement le démarrage et l'arrêt de l'enregistrement du programme (exactement comme pour une macro). Contrairement à son prédécesseur LIKE, EBP enregistre les actions de l'utilisateur au niveau de l'Adaptateur de Noyau Fonctionnel (et non pas au niveau des commandes de l'Adaptateur de Présentation). Cela a pour effet de rendre le programme enregistré indépendant de la couche de présentation (Figure 64).

Le problème majeur posé par la technique d'enregistrement des programmes dans EBP concerne l'intégration au sein du système même. Il s'agit de programmes générés que l'on peut appeler par des mécanismes interactifs spécifiques, mais qui ne sont jamais intégrés comme des composants natifs du système.

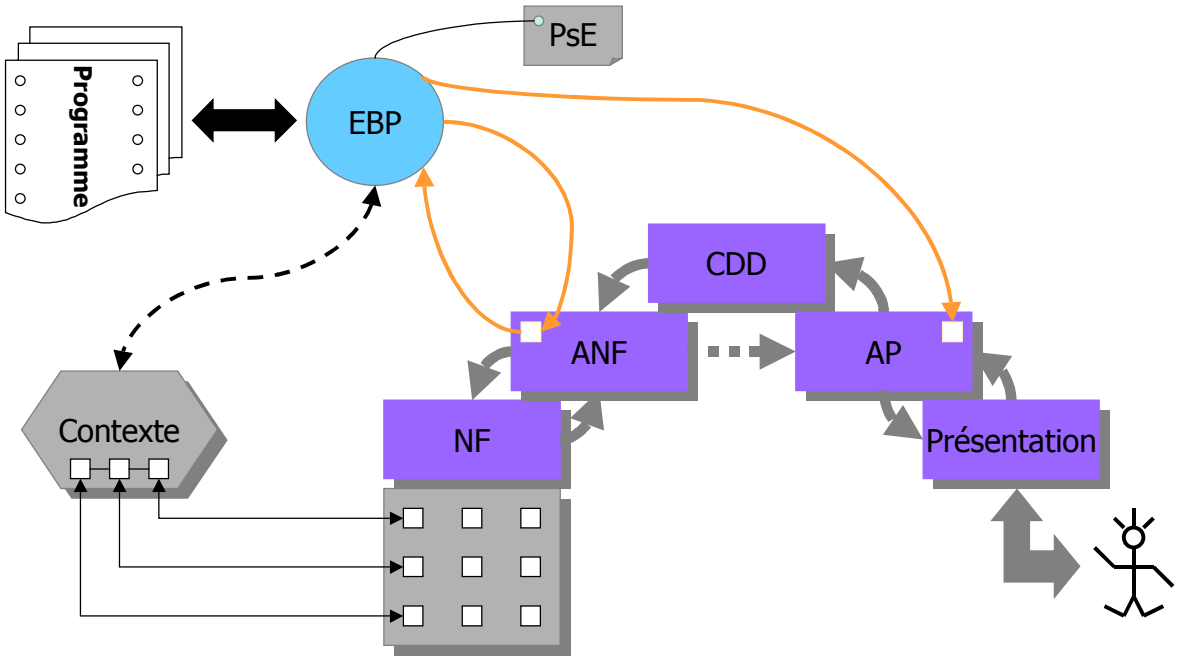


Figure 64: Architecture du système EBP

Généralisation de l'exemple

Pour généraliser l'exemple, EBP s'appuie sur la notion de contexte dynamique : chaque sous programme possède un contexte, utilisable par un sous programme appelant uniquement globalement.

Dans EBP :

- Les paramètres sont nommés par l'utilisateur.
- Les variables implicites sont numérotées de manière automatique.

Description de l'exemple

Dans EBP, l'utilisateur sait qu'il est en train de construire un programme (au contraire de certains autres systèmes de PsE). L'utilisateur signifie donc au système qu'il est prêt à commencer la description de l'exemple qui va permettre de créer le programme. Il commence par donner les paramètres du programme en les nommant et en leur assignant une valeur pour l'exemple en cours (le type des paramètres est déduit des valeurs saisies). Ensuite, il utilise les commandes de base du système pour créer la pièce désirée. Pour chaque fonction utilisée, il peut avoir recours aux paramètres (en utilisant explicitement les noms fournis au début) ou à des constantes. D'autre part, toutes les structures de contrôle d'un langage de programmation classique sont disponibles (conditions, boucles).

Structures de contrôles

Le contrôle du programme a lieu à l'intérieur du fichier texte, par l'intermédiaire de mots clefs reconnus (Figure 65).

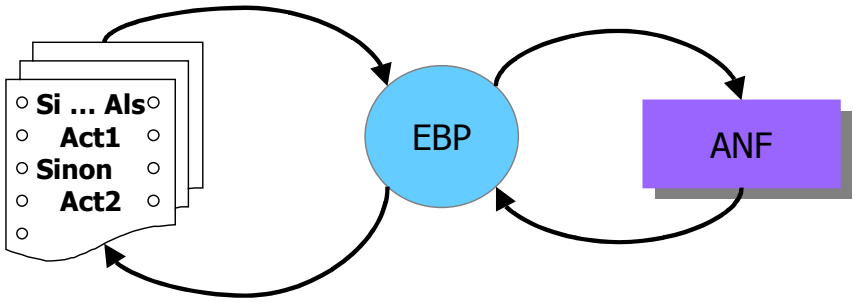


Figure 65 : Contrôle du programme dans EBP

Liens entre le système de PsE et le Noyau Fonctionnel

Dans EBP, le lien avec le Noyau Fonctionnel est beaucoup plus fort qu'il ne l'est dans LIKE puisque ce sont cette fois des commandes de l'ANF qui sont enregistrées.

Le niveau d'enregistrement des actions à ce niveau du dialogue permet de connaître la sémantique des actions appelées et donc de générer des programmes neutres, déconnectés complètement de toute présentation. D'un autre côté, le système d'enregistrement est de ce fait lié au Noyau Fonctionnel et n'est pas exploitable directement avec un nouveau modèle.

3.2.3 GIPSE

Description

GIPSE est un système de Programmation sur Exemple créé en 1999 par Guillaume Patry. Il s'agit d'un Enregistreur de macros évolués. Le but est l'intégration de macros dans le système (que les macros soient vues et utilisées comme des composants natifs du système). Le Contrôleur de Dialogue est capable d'appeler de la même façon des fonctions ayant un questionnaire très différent. La présentation inclut ces nouvelles fonctions.

A partir d'une application générique, l'utilisateur peut construire de nouvelles fonctions en se basant sur celles existant et les inclure dans la présentation de sa nouvelle application.

Enregistrement des programmes

Les programmes sont enregistrés de manière explicite : l'utilisateur commande explicitement le démarrage et l'arrêt de l'enregistrement du programme (exactement comme pour une macro) (Figure 66).

Généralisation de l'exemple

Comme dans EBP, les paramètres sont nommés explicitement par l'utilisateur, les variables implicites sont numérotées de manière automatique, …

Description de l'exemple

Encore une fois, on se place dans un contexte particulier : l'utilisateur sait pertinemment qu'il est en train d'écrire un programme. Le principe d'enregistrement reste donc toujours le même : l'utilisateur indique au programme qu'il va enregistrer un programme, nomme

les paramètres et exécute toutes les opérations de constructions correspondant à ce programme. Une fois terminé, il peut demander au système d'intégrer le programme en tant que véritable fonction.

Le système n'intègre pas de structures de contrôle.

Liens entre le système de PsE et le Noyau Fonctionnel

GIPSE ne connaît pas le Noyau Fonctionnel mais uniquement les fonctions du Contrôleur de Dialogue. Les programmes construits avec GIPSE ne sont pas consultables par l'utilisateur.

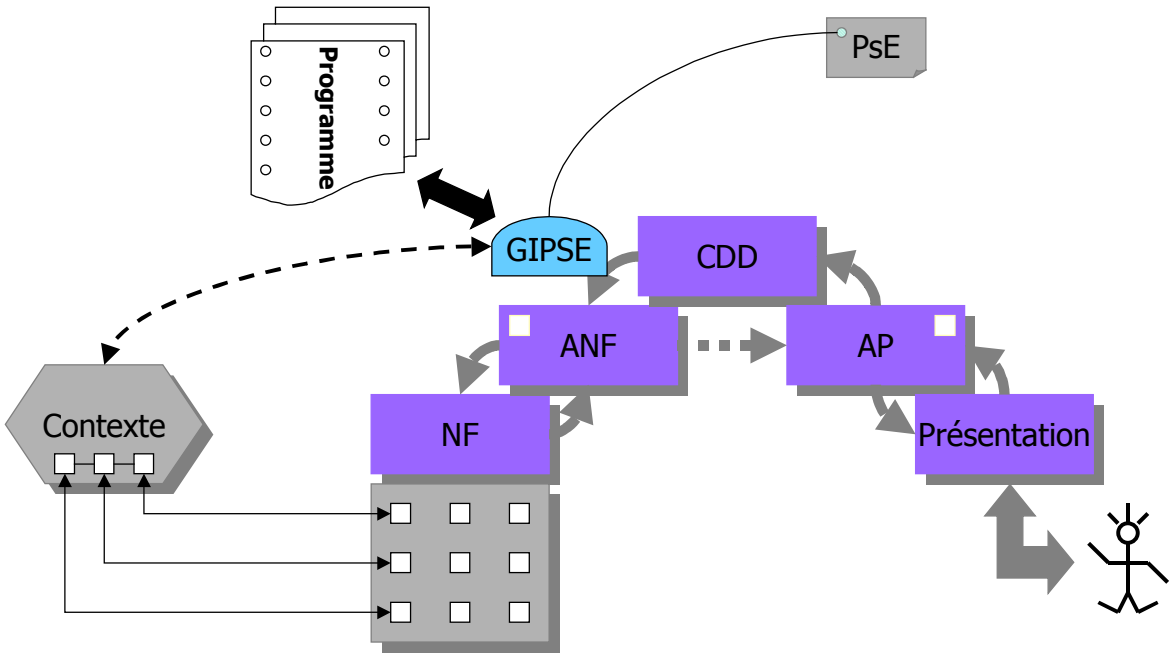


Figure 66 : Architecture du système GIPSE

3.2.4 TEXAO

Description

TEXAO est un système de PSE appliqué à la CAO, développé par Guillaume Texier en 2000. Il permet à un utilisateur « final » de construire de nouvelles classes (au sens objet du terme) de manière interactive.

Enregistrement des programmes

Dans TEXAO, l'utilisateur crée des classes d'objets. En ce sens, la tâche diffère un peu de celles vues précédemment. Pour définir une classe, l'utilisateur doit définir un constructeur puis des attributs.

La définition du constructeur de la classe se passe exactement comme dans les autres systèmes. L'utilisateur doit d'abord saisir les paramètres du programme. A partir de là, ses actions sont enregistrées sous la forme d'un DAG (Direct Acyclic Graph), dans lequel on différencie, entre les nœuds, les paramètres des constantes. L'arbre de construction, qui correspond à une instance de la classe construite, est ensuite généralisé.

A partir d'une classe ainsi construite, on peut définir des attributs dérivés. Ceux-ci sont des attributs calculés à partir d'attributs déjà définis dans la classe (Figure 67).

Généralisation de l'exemple

Pour généraliser l'exemple décrit, TEXAO utilise un certain nombre de règles définies sur les DAG : les paramètres sont clairement identifiés dès le départ par l'utilisateur, les constantes sont toutes les autres valeurs des contenues dans des feuilles du DAG, les variables internes sont toutes les valeurs contenues dans des nœuds internes du DAG.

Les classes d'objet ainsi construites sont intégrées au même titre que les classes de base à l'intérieur de l'application : en effet, le système est capable, à partir du DAG d'une instance généralisé, de générer une classe dans un langage de programmation comme le C++. Le constructeur de cette classe est ensuite relié à la présentation par l'intermédiaire d'un bouton.

Description de l'exemple

Le principe de construction reste toujours le même : l'utilisateur décrit son exemple grâce à des menus, en utilisant les classes de base du système.

Le système ne permet pas l'utilisation de structures de contrôle.

Liens entre le système de PSE et le N.F.

Comme dans EBP, l'enregistrement des programmes se fait au niveau de l'Adaptateur de Noyau Fonctionnel. Cette connaissance beaucoup plus grande de la sémantique des actions entraîne une plus grande généralité des programmes. Mais cela implique également le fait que le système de PSE est fortement lié au modèle et ne peut donc pas être transféré facilement dans un autre contexte.

Structures de contrôle

Les boucles et les conditions ne sont pas implémentées. Le programme se déroule en suivant la structure du modèle intermédiaire (le modèle propriétaire de représentation des classes créées interactivement).

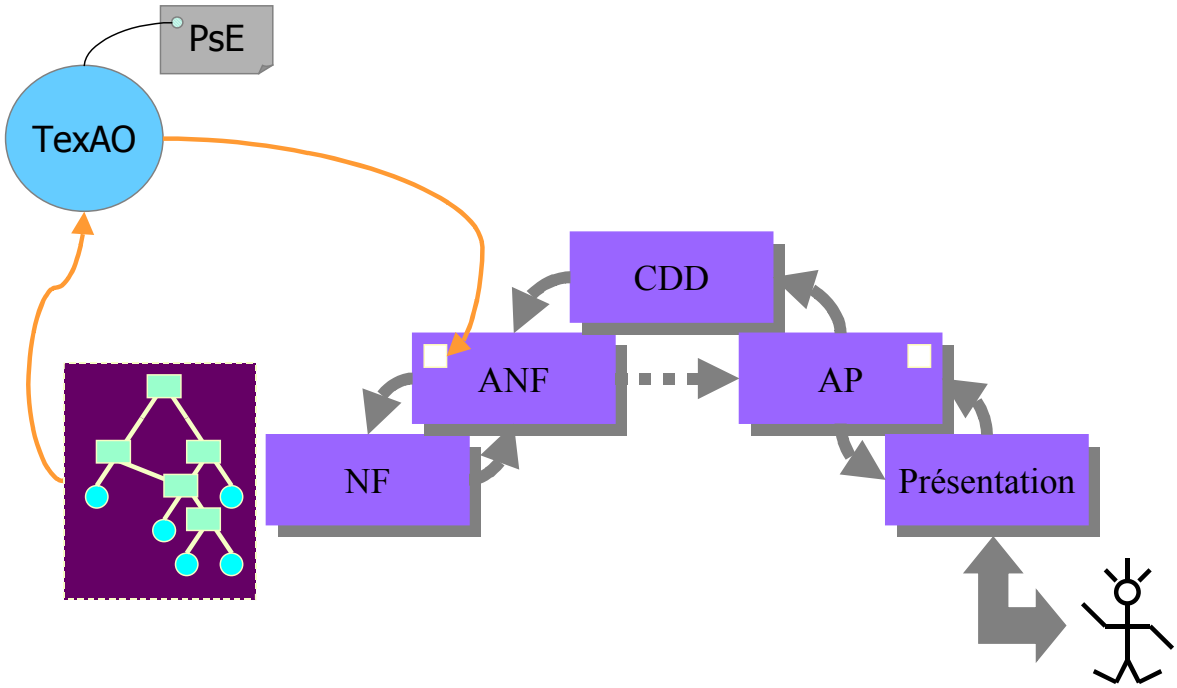


Figure 67 : Architecture du système TexAO

3.2.5 *Conclusion partielle*

Les systèmes de PsE développés au LISI ont tous pour point commun la description interactive de programmes. Ils reposent tous sur la même architecture logicielle (H[4]), dont le mécanisme du contrôleur de dialogue se prête idéalement aux dialogues structurés (or ces dialogues structurés sont fondamentaux dans le cadre de la CAO). La description des programmes se déroule toujours de la même façon : on commence par identifier clairement les paramètres du programme, puis on décrit le programme. La description du programme consiste toujours à utiliser les fonctions de base du modèle avec des constantes ou les valeurs des paramètres fournis au début de la phase d'enregistrement. Toutes les autres valeurs, calculées en cours de route, sont elles considérées comme des variables internes, que l'utilisateur n'a pas besoin de nommer.

Dans tous ces systèmes, on voit deux approches se dégager : une première consiste à enregistrer les actions au niveau de l'interface, l'autre à les enregistrer au niveau du Noyau Fonctionnel. L'avantage d'être plus proche de l'interface est que le système de PSE peut-être facilement réutilisé sur une application totalement différente (le système d'enregistrement ne connaît rien de la sémantique des actions). Par contre, tout changement dans l'interface invalide les programmes enregistrés. A l'inverse, les systèmes enregistrés au niveau du NF ont l'avantage d'être plus généraux (le fait de connaître la sémantique des actions permet une plus grande abstraction des fonctions enregistrées) et donc d'être ré-exécutable dans des environnements différents. Par contre, le système d'enregistrement, lui, est lié à l'architecture du système et ne peut donc pas être utilisé avec un modèle différent.

Cette analyse préalable nous permet de nous rendre compte des caractéristiques que notre environnement peut emprunter aux systèmes existants.

A un niveau interactif d'abord, on se place dans un cadre identique à celui de EBP. L'utilisateur du système, bien que n'étant pas un programmeur averti, sait pertinemment qu'il est en train de créer un programme. De ce fait, on peut prendre des solutions tournées vers une utilisation explicite de certains concepts : les structures de contrôle peuvent être appelées par l'intermédiaire de menus et les paramètres du programme peuvent être déclarés puis utilisés par la suite en toute connaissance de cause.

En ce qui concerne la Programmation sur Exemple elle-même, l'enregistrement des actions peut se faire au niveau de l'Adaptateur de Noyau Fonctionnel. En agissant de la sorte, on assure une indépendance entre le programme créé et la Présentation. Nous verrons plus précisément au cours de la description de notre environnement pourquoi il s'agit de la meilleure voie à utiliser.

Ceci étant, le cadre d'application auquel nous nous intéressons présente des différences notables avec celui des systèmes étudiés. Il exige de ce fait des solutions originales pour un certain nombre de problème.

Tout d'abord, nous nous plaçons dans un contexte où l'exécution du programme est associé à la structure des objets : le contrôle global est ainsi intimement lié au parcours de la structure puisque des « bouts » du programme sont adjoints à différents éléments du modèle. D'autre part, nous voulons permettre la modification du programme en cours d'exécution et ne pas faire de différences, comme cela est fait dans les systèmes étudiés, entre les phases de conception et d'exécution.

Pour faire un parallèle avec les autres systèmes, l'architecture finale d'une application créée avec notre environnement ressemblerait à celle présentée sur la Figure 68. On voit sur ce schéma que, contrairement aux systèmes étudiés, le programme est intégré dans la structure de données : certains éléments du modèles « portent » des morceaux du programme (appelés scénarii). L'exécution du programme dans son intégralité consiste à parcourir la structure et à appliquer les différents comportements rencontrés. Contrairement à un système comme TexAO, totalement paramétrique, nous avons besoin ici de posséder une représentation externe des comportements associés à telle ou telle partie du modèle.

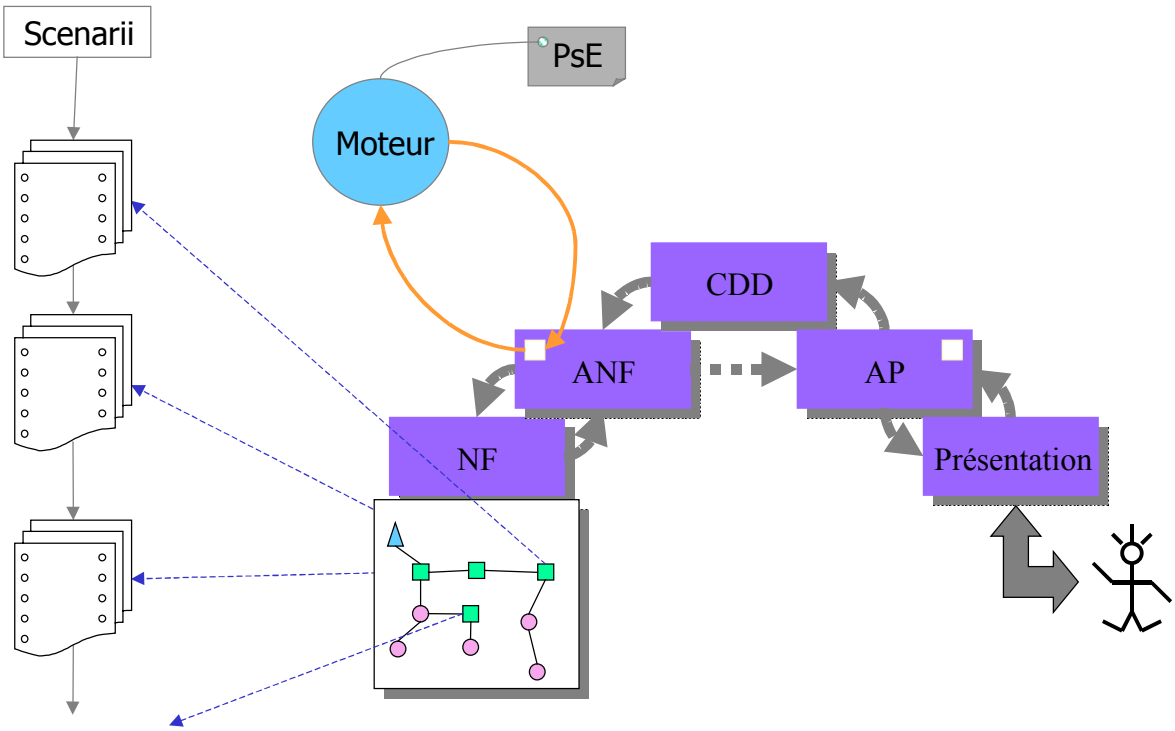


Figure 68 : Architecture d'une application créée avec notre environnement

La suite de cette partie présente les deux éléments de notre environnement : le modèle de Noyau Fonctionnel générique et le moteur de Programmation sur Exemple sur lequel il s'appuie.

3.3 Un modèle de Noyau Fonctionnel pour la création de métamorphoses

Le premier chapitre de ce manuscrit nous a éclairé sur les techniques utilisées en animation et notamment en simulation de métamorphoses. Nous avons qu'une métamorphose était une suite d'objets $O_1 \ldots O_n$, O_i représentant l'objet à l'instant i, dans laquelle chaque objet peut se déduire du précédent en appliquant un ensemble d'opérations définies sur lui. Ces objets peuvent être structurés et chaque sous-objet peut lui-même posséder un comportement propre. Ainsi, métamorphoser un objet revient à appliquer tous les comportements définis sur lui et sur ses sous-éléments dans un ordre donné. Cette technique permet de pouvoir contrôler les transformations d'un objet donné à un niveau plus ou moins global.

Pour pouvoir créer un environnement de programmation sur exemple capable de fonctionner sur tout modèle permettant de représenter les évolutions d'un objet naturel structuré, nous avons besoin de définir un modèle générique sur lequel il pourra s'appuyer. Pour cette raison, nous avons abstrait les caractéristiques communes des modèles d'objets naturels structurés.

3.3.1 Structure

3.3.1.1 Supports de scénario et de programme

Selon nous, les aspects fondamentaux des modèles représentant des objets naturels structurés sont :

- Leur décomposition en sous-objet ;
- Le lien de certains éléments de la structure à des « scénarii » représentant leur comportement ;
- Le parcours de l'objet pour l'application des « scénarii » dans un ordre donné.

Nous avons donc décomposé un objet naturel structuré en trois éléments.

Une classe abstraite est ainsi chargée de représenter les « supports de scénario ». Il s'agit d'identifier les éléments de la structure de données qui posséderont un lien vers un scénario de croissance. Par exemple, dans une application représentant des arbres en 1D½, les axes seront des supports de scénario.

Une classe générique est chargée de représenter le « support de programme ». Il s'agit d'identifier, dans la structure de données, le point de départ du parcours de la structure. Cette classe définit donc des méthodes permettant le parcours de l'objet ainsi que l'exécution des scénarii associés aux différents « supports de scénarii ».

Une dernière classe représente les « scénarii ». Pour l'instant, il nous suffit de savoir qu'elle fait le lien entre le modèle et le moteur de PsE. Nous verrons dans un deuxième temps sa composition.

La Figure 69 présente le schéma UML de ce modèle.

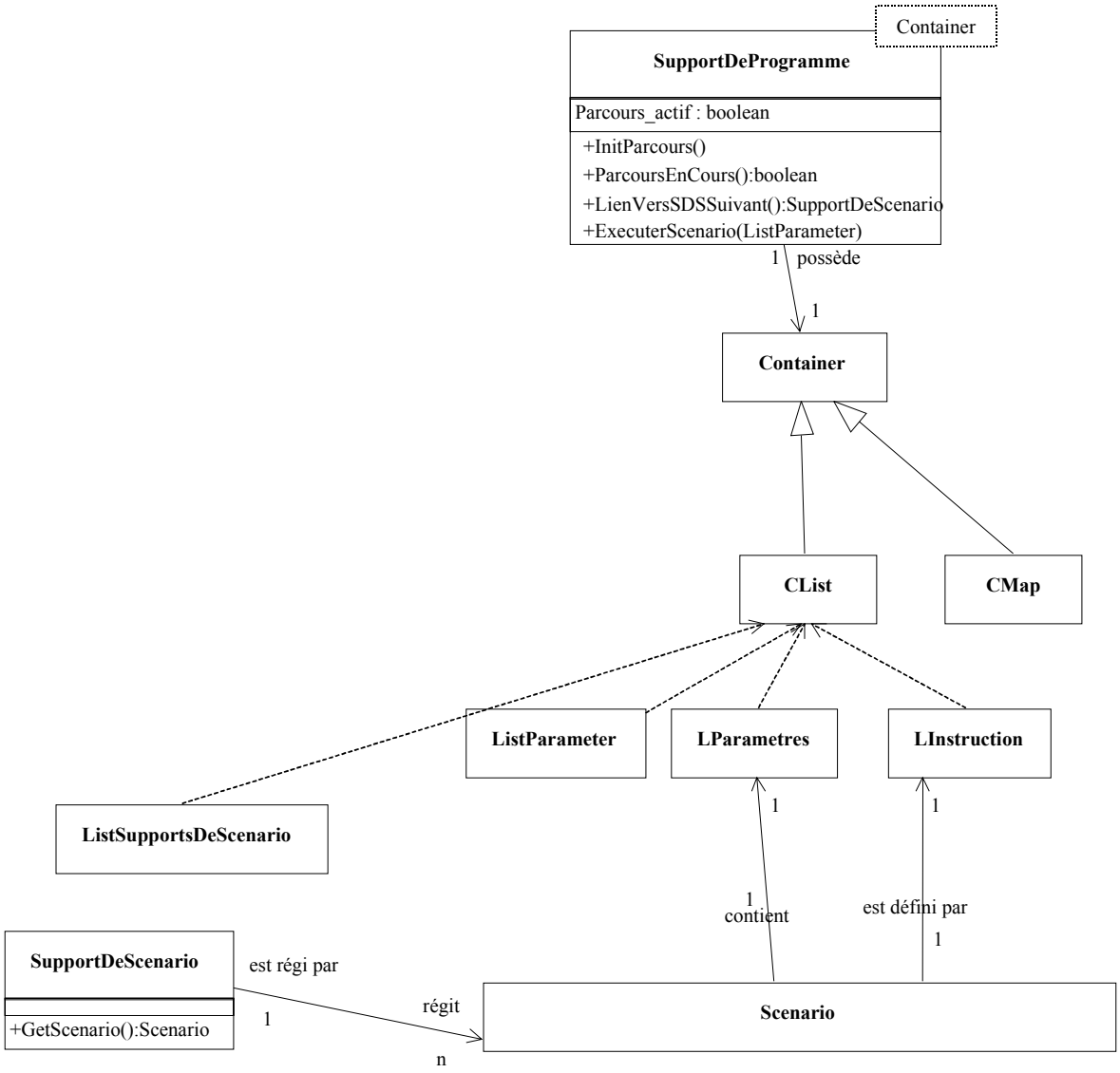


Figure 69 : Schéma UML du modèle générique représentant un objet naturel structuré

3.3.2 Utilisation

Pour utiliser ce modèle, le concepteur d'une application doit simplement greffer son propre modèle sur celui-ci. Cela requiert donc de sa part une identification préalable des composants susceptibles de pouvoir représenter d'une part les supports de scénario, d'autre part le support de programme. On peut ici noter qu'il peut y avoir plusieurs supports de scénario. Certains modèles possèdent en effet plusieurs composants distincts supportant une part du scénario de croissance global.

La marche à suivre consiste donc à instancier la classe générique « support de programme » avec la classe du modèle correspondante puis à spécifier, pour chaque élément du modèle identifié comme tel, que la classe dérive de « support de scénario ».

3.4 Moteur de Programmation sur Exemple

A partir de ce modèle générique, nous avons établi un moteur de Programmation sur Exemple. Ce terme général regroupe un ensemble de composants :

- Une structure liée au modèle précédemment décrit représentant un « scénario ». Celle-ci est attachée plus précisément aux « supports de scénario » et stocke les actions généralisées de l'utilisateur représentant le comportement d'un composant ;
- Un interpréteur, capable de ré-exécuter un scénario donné en exécutant les actions du Noyau Fonctionnel correspondantes ;
- Un cadre de conception d'enregistrement de programmes, qui se présente sous la forme de règles que doit suivre le concepteur de l'application pour permettre l'enregistrement des actions de l'utilisateur.

3.4.1 H⁴ : un interpréteur de langage

Considérons l'exemple de la croissance d'un arbre filaire (Figure 70). La métamorphose de cet arbre est exprimable de manière algorithmique. Le pseudo-code de la Figure 71 présente ainsi ces transformations et illustre le fait que la description d'une métamorphose utilise toutes les notions classiques de la programmation fonctionnelle : variables, appels de fonctions à l'intérieur d'autres fonctions, boucles, conditions, etc. Ce que nous montre en outre cet exemple, c'est que comprendre le scénario de croissance d'un objet revient justement à interpréter un langage de programmation fonctionnelle. Il faut en effet être capable de reconnaître les commandes, les paramètres, etc.

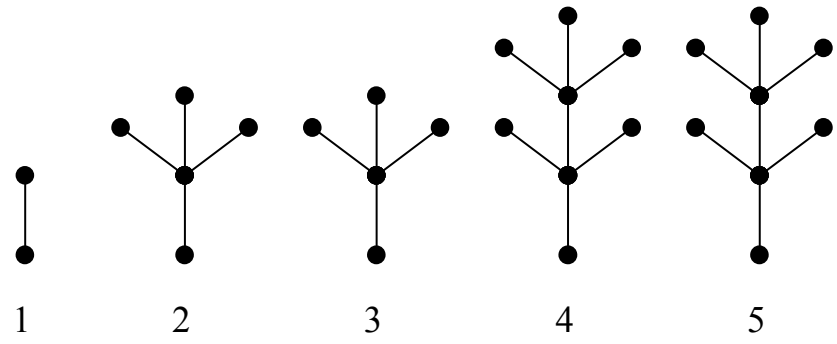
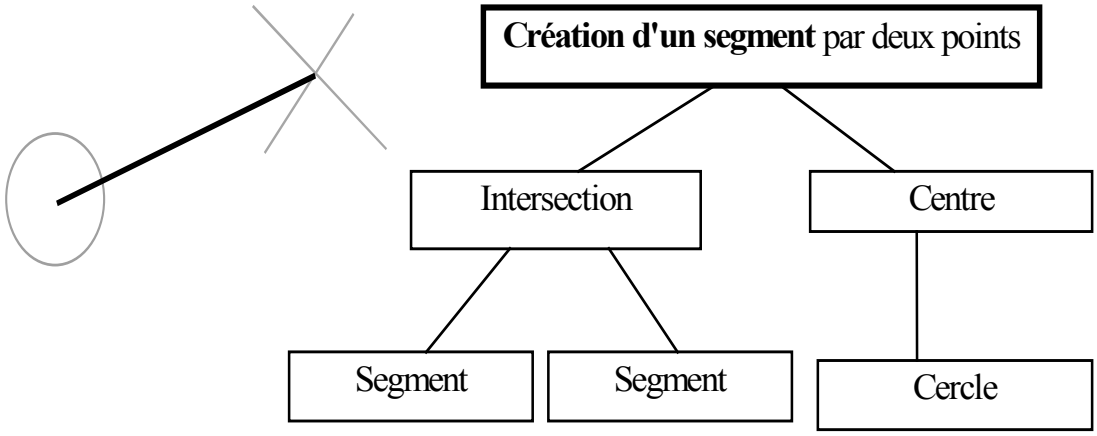


Figure 70 : Exemple de croissance d'arbre en 1D½ et pseudo-algorithme de construction associé

```
axePrinc = creerAxePrincipal(long, angle, pos)
Pour chaque top horloge t Faire
    Si (t modulo 2 = 0) Alors
        insererAxe (dernierEN(axePrinc), long, angle)
        insererAxe (dernierEN(axePrinc), long, -angle)
        ajouterEN (axePrinc, longueur(dernierEN(axePrinc)), angle)
    FinSi
FinPour
```

Figure 71 : Pseudo-algorithme correspondant à la croissance de la Figure 70

Le fonctionnement du contrôleur de dialogue de H4 agit comme un interpréteur de langage fonctionnel dans lequel l'alphabet serait constitué des jetons et commandes et l'analyseur syntaxique serait le couple Interacteurs / Moniteur. Un tel interpréteur permet de reconnaître des phrases telles que celle décrite sur la Figure 72.




```
CréerSegment(Intersection (S1, S2), Centre (C1))
```

Figure 72 : Exemple de phrase reconnue par le Contrôleur de Dialogue de H4

L'architecture de H4 propose un fonctionnement qui permet dans un sens plus général de reconnaître un langage fonctionnel bien défini. On peut alors faire le parallèle avec l'exemple de la Figure 70 : on manipule en effet des fonctions qui font elles-mêmes appel à d'autres fonctions. On crée par exemple un entre-nœud à l'extrémité d'un axe, dont la longueur est la même que celle du dernier entre-nœud. La programmation d'une métamorphose d'objet naturel structuré, comme par exemple la croissance d'un arbre, via un système niveau « animateur » revient également à reconnaître un langage de programmation fonctionnel. Il s'agit d'un sous-ensemble de la programmation manipulant des structures de contrôle, des constantes, des variables, des paramètres et des appels de fonctions comme paramètres d'autres fonctions. Comme nous l'avons vu dans le chapitre 2, c'est justement ce sous ensemble de la programmation que le contrôleur de dialogue de H4 est capable de d'interpréter.

Notre idée consiste à utiliser le fonctionnement du contrôleur de dialogue de H4 comme interpréteur des différents scénarios constituant le programme de la métamorphose.

3.4.2 Interpréteur de scénario

Le principe sur lequel nous nous basons est le suivant : un scénario est composé d'une suite ordonnée de jetons, au sens H4 du terme. Ainsi, une séquence de cette liste est toujours formée de : un jeton commande, correspondant à une action du système, suivie d'un certain nombre de jetons paramètres, portant les valeurs des différents paramètres de l'action. Les jetons d'un scénario sont passés, dans l'ordre, au Moniteur de Scénario. Celui-ci les propose successivement à une hiérarchie d'interacteurs composés de questionnaires. Exactement comme dans le cas du Contrôleur de Dialogue, les interacteurs stoc-

kent les jetons qu'ils acceptent et fournissent éventuellement des valeurs, proposés ensuite aux interacteurs placés en amont dans la hiérarchie. Lorsqu'un questionnaire a reçu toutes les informations nécessaires à son activation, il transmet les valeurs à la méthode correspondante de l'Adaptateur de Scénario. Celle-ci se charge d'effectuer les transformations de type nécessaire afin de pouvoir à son tour appeler la méthode adaptée du Noyau Fonctionnel. La Figure 73 illustre l'architecture de ce Contrôleur de Scénario.

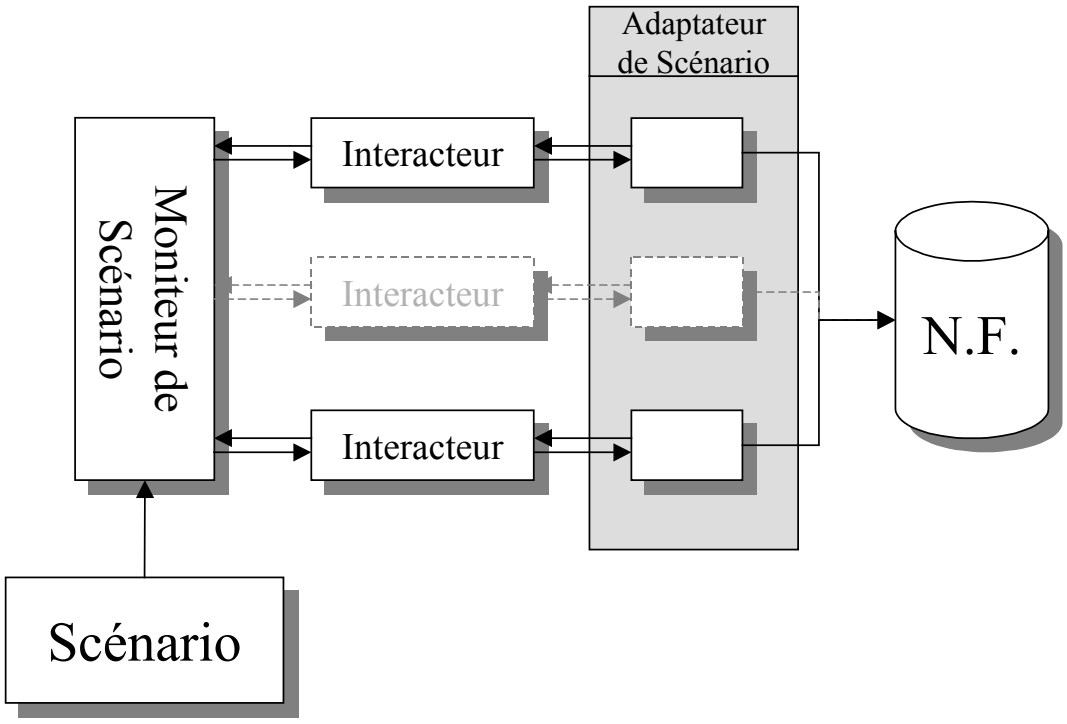


Figure 73 : Architecture de l'interpréteur de scénario

On peut voir ce fonctionnement comme celui d'un contrôleur de dialogue dans lequel les entrées ne seraient pas fournies par la couche de Présentation, et donc l'utilisateur, mais par un flux continu préenregistré : le scénario.

3.4.2.1 Composants

Nous décrivons ici les différents éléments composant l'architecture du moteur de programmation sur exemple. La Figure 74 récapitule graphiquement dans un format UML leur organisation entre eux ainsi que par rapport au modèle générique de représentation d'un objet naturel structuré.

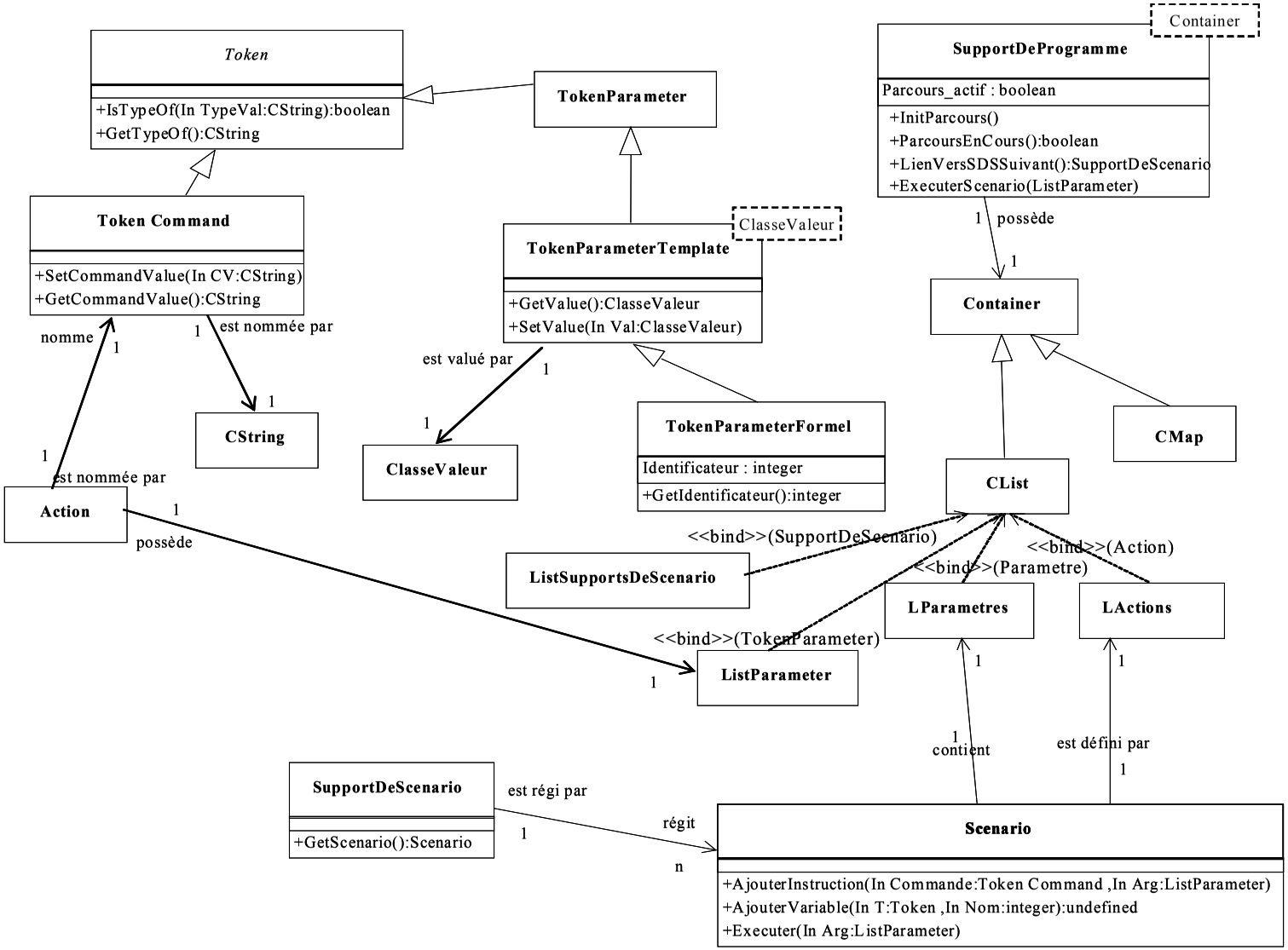


Figure 74 : Représentation UML des classes constituant l'interpréteur de scénario, les scénarii et le modèle générique d'objets naturels structurés

3.4.2.1.1 Scénario

La structure de Scénario fait le lien entre l'interpréteur (le Contrôleur de Scénario) et le modèle générique. Nous avons effectivement vu précédemment que le concepteur devait, dans son noyau fonctionnel, identifier les éléments « porteurs » d'un morceau du comportement général de ses objets. Ceux-ci doivent alors être mis en correspondance avec la structure prédéfinie de « Support de Scénario » qui possède un lien vers un Scénario.

Un Scénario est constitué d'une suite ordonnée de jetons. Cette séquence débute toujours par une commande qui permet d'initier le questionnaire correspondant à l'action. Elle est immédiatement suivie des paramètres utilisés par le questionnaire. Evidemment, exactement comme dans le cas du Contrôleur de Dialogue, une séquence « Commande – Paramètres » peut s'intercaler pour fournir éventuellement une valeur à l'action préalablement initiée.

L'environnement gère une liste de scénarii, correspondant aux scénarii des différents éléments de l'objet dont on représente la croissance.

3.4.2.1.2 Jetons

Les jetons, comme dans le cadre du Contrôleur de Dialogue, doivent être définis par le concepteur en fonction des types de données du Noyau Fonctionnel.

D'autre part, à la notion classique de jetons, nous avons ajouté celle de jeton « paramètre formel ». Ceux-ci dérivent de la classe « jeton paramètre » mais possède en plus un nom permettant de les identifier. Cette nouvelle définition est rendue indispensable par le fait que l'on veut pouvoir paramétrer les scénarii et modifier, à l'appel, la valeur de certains d'entre eux. Le mécanisme de nomination nous permet de retrouver, dans un scénario, un paramètre donné et de lui affecter une valeur différente pour toute nouvelle exécution.

3.4.2.1.3 Moniteur de Scénario

Le Moniteur de Scénario a un rôle absolument identique à celui du Moniteur « classique ». Il reçoit des jetons en provenance d'un flux (le scénario remplaçant ici l'utilisateur) et il les transmet aux différents interacteurs, dans l'ordre de la hiérarchie définie par le concepteur. Lorsqu'un jeton est refusé par un interacteur, il est rendu au Moniteur qui se charge de le proposer à l'interacteur suivant.

3.4.2.1.4 Interacteurs et Questionnaires

Les questionnaires, comme dans le cadre du contrôleur de dialogue, sont chargés de faire le lien entre le Contrôleur de Scénario et l'Adaptateur de Scénario. Ils sont définis par une signature et sont chargés d'activer une méthode de l'Adaptateur de Scénario dès qu'ils ont reçu les paramètres nécessaires. Les questionnaires de même niveau d'abstraction sont regroupés au sein d'interacteurs identiques. Ceux-ci sont représentés par des automates et sont ainsi chargés de distribuer les jetons proposés par le Moniteur aux différents

questionnaires qui les composent, en fonction de l'état dans lequel il se trouve (donc en fonction des jetons déjà reçus et stockés).

3.4.2.1.5 Adaptateur de Scénario

L'Adaptateur de Scénario correspond, dans le Contrôleur de Dialogue, à l'Adaptateur de Noyau Fonctionnel. C'est ici que se fait le lien entre les données et les méthodes du scénario et celles du modèle du Noyau Fonctionnel. Ainsi, à chaque questionnaire défini correspond une méthode de l'Adaptateur de Scénario qui transforme les jetons en données du Noyau Fonctionnel et appelle la primitive correspondante.

3.4.2.2 Structures de contrôle

Pour modéliser les Structures de Contrôle, il suffit d'ajouter les interacteurs adaptés dans la hiérarchie du Moniteur de Scénario et de les associer à un Filtre permettant de traiter les jetons provenant du scénario en conséquence.

3.4.2.2.1 Structure conditionnelle

Prendre en compte les structures conditionnelles (Si Alors Sinon) revient à permettre l'évaluation d'un prédicat (la condition) et à bloquer certains jetons en fonction du résultat de cette évaluation.

L'évaluation d'un prédicat est rendue très facile par le fonctionnement du Contrôleur de Scénario. Il suffit en effet de définir les questionnaires des fonctions logiques et de les lier à des méthodes du Noyau Fonctionnel capables d'en calculer effectivement le résultat. Les questionnaires doivent ensuite être regroupés à l'intérieur d'un même interacteur. Le fonctionnement du Contrôleur fait le reste, permettant notamment de fournir les paramètres dans un ordre quelconque.

Mettre en place une structure conditionnelle revient à bloquer certains jetons selon le résultat de l'évaluation d'un prédicat. Pour bloquer les jetons indésirables (ceux de la branche que l'on ne veut pas exécuter), une solution consiste à associer des interacteurs à un filtre placé entre le Moniteur de Scénario et le Scénario lui-même.

On place ainsi dans la hiérarchie deux nouveaux interacteurs représentant respectivement le « Si », et les « Alors » et « Sinon » de la condition (Figure 75). Le filtre est alors chargé, selon l'état dans lequel se trouve ces interacteurs et les jetons qu'il reçoit du scénario, de bloquer ou de laisser passer les informations obtenues.

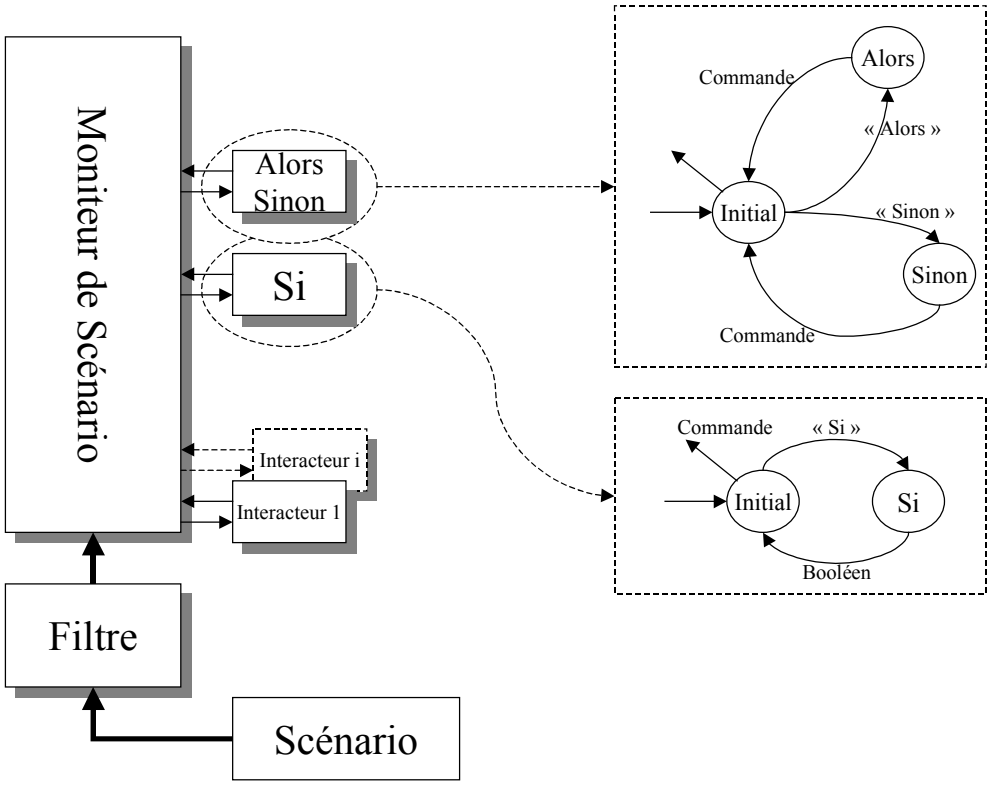


Figure 75 : Modélisation d'une structure conditionnelle

Dans un scénario, une condition est ainsi représentée par une séquence de jetons du type :

[« Si » – Séquence de jetons – « Alors » – Séquence de jetons – « Sinon » - Séquence de jetons]

L'interacteur de condition « Si » est chargé d'activer l'interacteur correspondant au résultat de l'évaluation du prédicat. Ainsi, lorsque tous les jetons correspondant à ce prédicat ont été transmis au Moniteur, une valeur booléenne est retournée à celui-ci qui la passe à l'interacteur de condition. Selon sa valeur, VRAI ou FAUX, l'interacteur retourne un jeton « commande » de valeur « Alors » ou « Sinon », permettant l'activation du questionnaire correspondant à l'intérieur de l'interacteur « AlorsSinon ».

Le Filtre analyse les entrées proposées par le scénario et bloque les jetons en fonctions de la branche conditionnelle dans laquelle ils se trouvent. Son fonctionnement repose sur une pile d'état, nécessaire pour traiter les cas de conditions imbriquées.

A l'arrivée d'un jeton commande « Si », le Filtre laisse passer les jetons. La commande est transmise à l'interacteur « Si » qui se place en position d'attente d'un booléen. Les jetons suivant sont utilisés par des interacteurs pour calculer et fournir au Moniteur une valeur booléenne. Celle-ci est transmise à l'interacteur « Si » qui retourne selon les cas une commande « Alors » (pour un booléen valant VRAI) ou « Sinon » (pour un booléen valant FAUX). On se trouve alors dans une configuration où :

- L'interacteur « Si » est de nouveau dans son état initial ;

- Un et un seul des questionnaires « Alors » et « Sinon » est activé ;
- Le Filtre reçoit un jeton commande « Alors » ;

Le Filtre teste alors l'état de l'interacteur « AlorsSinon ». Si celui-ci se trouve dans l'état « Alors », il empile la commande « Alors » sinon, il empile la commande « Sinon ».

La gestion des jetons est ensuite assurée par les règles suivantes :

- A l'arrivée d'une commande « Si », on laisse passer les jetons jusqu'à la commande « Alors » suivante ;
- A l'arrivée d'une commande « Alors », on empile « Alors » dans la pile d'état du Filtre si l'interacteur « AlorsSinon » se trouve dans l'état « Alors ». Dans l'autre cas, on empile la commande « Sinon » ;
- A l'arrivée d'une commande « Alors » ou « Sinon », on compare la commande reçue et celle située au sommet de la pile. En cas d'égalité, on laisse passer les jetons, sinon, on les bloque ;
- A l'arrivée d'une commande « FinSi », on dépile.

L'application de ces règles pose néanmoins un problème lorsque l'on parcours une branche dont la condition n'est pas vérifiée. En effet, les règles ci-dessus nous demandent d'en bloquer les instructions tout en continuant à les examiner. Or si cette branche contient elle-même une structure conditionnelle, on retombe inéluctablement sur un « Alors » ou un « Sinon » qui ne correspond pas à la structure conditionnelle initiale.

Pour éviter ce problème, il suffit, lorsque l'on se trouve dans un mode où l'on bloque les jetons, d'empiler les commandes « Si » que l'on rencontre, et de ne les dépiler que lorsque l'on se voit proposer une commande « FinSi ». De cette manière, la commande du sommet de la pile ne peut pas être égale à une commande « Alors » ou « Sinon », et tous les jetons correspondants à la structure conditionnelle incluse dans la branche à ne pas exécuter sont ignorés.

Remarque. *Les interacteurs permettant l'évaluation d'un prédicat doivent être placés SOUS les interacteurs chargés des conditions. De même, l'interacteur « AlorsSinon » doit être disposé au dessus de l'interacteur « Si ».*

3.4.2.2.2 Structure de répétition

De la même façon que pour les conditions, il est possible de simuler une structure répétitive de type « **Tant Que** Condition **Faire** … **FinTantQue** », en associant un Filtre, placé entre le Moniteur et le Scénario, et deux nouveaux interacteurs.

Une différence notable dans la façon de procéder vient du fait que l'on décompose la gestion de la répétition en deux phases distinctes. Une première étape permet d'enregistrer la condition de répétition ainsi que les différentes instructions de la boucle. La deuxième phase consiste à exécuter les instructions selon le résultat de l'évaluation de la condition.

Afin de pouvoir évaluer les conditions et rejouer les instructions d'éventuelles boucles imbriquées, le Filtre doit gérer deux listes. Une liste de conditions et une liste d'instructions (notons que les instructions et les conditions sont représentées de la même

façon, à savoir des séquences de jetons). Ces composants permettent de séparer une imbrication de boucles en différents éléments. Ainsi, le premier élément de la liste de conditions correspond à la séquence de jetons représentant la condition de la première boucle, le second à la condition de l'éventuelle deuxième boucle rencontrée séquentiellement. Il en est de même pour la liste d'instructions.

Enregistrement

Lorsque le Filtre se voit proposer une commande « TantQue » par le Scénario, il met en marche le processus d'enregistrement.

- Tant qu'il n'y a pas de jeton commande « Faire », les jetons sont conservés dans le premier élément de la liste des conditions ;
- Lorsque la commande « Faire » a été reconnue, les jetons suivants sont enregistrés dans le premier élément de la liste des instructions jusqu'à ce que le Scénario fournisse un jeton commande « FinTantQue » ;

Si un jeton commande « TantQue » se présente pendant la phase d'enregistrement, on effectue les mêmes opérations, mais en utilisant le premier emplacement libre des listes de conditions et d'instructions. Afin de savoir à quel « TantQue » se rapporte un « FinTantQue », le Filtre gère une pile d'état : il empile lorsqu'un « TantQue » est présenté, il dépile lorsqu'un « FinTantQue » survient.

La phase d'enregistrement est terminée lorsque la pile gérée par le Filtre est vide.

Exécution

La phase d'exécution repose sur le même principe que pour l'expression des conditions vue antérieurement. Deux interacteurs sont ajoutés à la hiérarchie du Moniteur de Scénario. Le premier, « TantQue », possède une transition sur un booléen : selon la valeur qui lui est fournie, l'interacteur retourne la commande « Faire » ou « Fin », qui est rendue au Moniteur. Celui-ci la transmet à l'interacteur placé immédiatement après : « Faire », dont l'état passe, selon le cas, à « Faire » ou « Fin » (Figure 76).

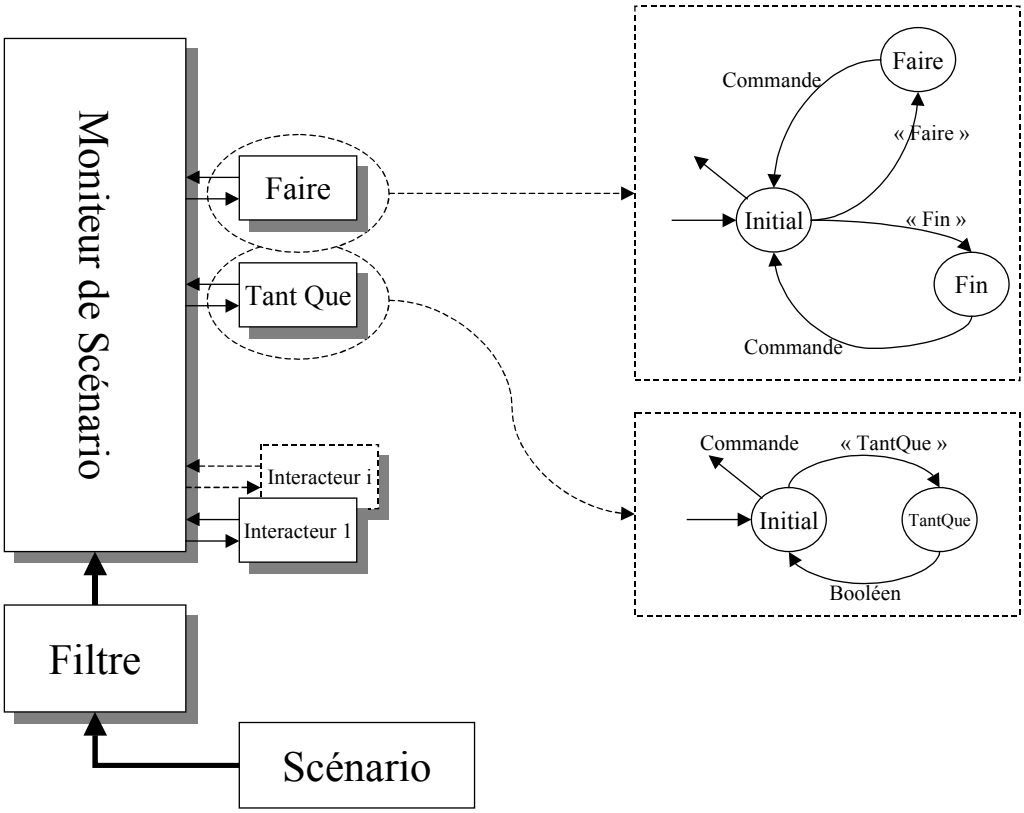


Figure 76 : Modélisation d'une structure de répétition

Lors de l'exécution de la boucle enregistrée, le Filtre fournit dans un premier temps au moniteur la condition du premier élément de la liste de conditions. Celle-ci est utilisée par le Moniteur pour approvisionner les interacteurs appropriés et produire une valeur booléenne utilisable par l'interacteur « TantQue », avec, comme nous venons de le voir, pour conséquence finale le changement d'état de l'interacteur « Faire ». Le filtre analyse l'état de cet interacteur et selon qu'il se trouve dans un état « Faire » ou « Fin » exécute ou non les instructions de la boucle.

La gestion des boucles imbriquées implique l'analyse de quelques cas supplémentaires :

- Si une commande s'avère être un « TantQue » (ce qui signifie que l'on est en présence d'une boucle imbriquée), le Filtre exécute alors la condition suivante dans la liste des conditions puis fait de même avec les instructions ;
- Si une commande est identifiée comme étant un « FinTantQue », le Filtre exécute à nouveau la condition courante de la liste de conditions. Il analyse alors de nouveau l'état de l'interacteur « Faire » pour savoir si oui ou non il doit exécuter les instructions correspondantes. Dans le cas où ce n'est pas possible, le Filtre reprend l'exécution des instructions de la liste précédente à l'endroit où il s'était arrêté.

3.4.2.2.3 Conclusion

Il est ainsi possible de simuler les structures de contrôle traditionnelles que sont les conditions et les boucles en utilisant le système du Contrôleur de Dialogue de H[4]. Mal-

heureusement, l'introduction du filtre et l'identification de plusieurs interacteurs parmi les autres rompt l'harmonie du système et n'assure plus la propriété selon laquelle le Moniteur ne connaît rien des interacteurs en dehors de leur positionnement à l'intérieur de sa hiérarchie. Cependant, exactement comme pour le cas de la Manipulation Directe, les modifications apportées n'influent pas sur le fonctionnement « classique » du Moniteur.

3.4.3 Enregistrement des scénarii

L'environnement que nous avons mis en place ne permet pas d'automatiser les actions d'enregistrement. Nous sommes partis dès le départ dans l'idée de réaliser un environnement permettant à un concepteur de mettre en place facilement des concepts de PsE à l'intérieur de son application. Nous avons considéré que ce concepteur était déjà en possession d'un Noyau Fonctionnel permettant de modéliser des objets naturels structurés et de les transformer par l'intermédiaire d'un certain nombre de primitives. Mais nous avons également considéré que ce modèle comportait déjà ses propres primitives d'affichage et que le concepteur pouvait aussi bien vouloir travailler en OpenGl qu'en DirectX, développer son interface en Java/Swing ou avec les MFC de Visual C++, etc. Pour lui laisser un maximum de liberté, nous avons donc travaillé uniquement sur la partie indépendante de l'interface graphique.

Malheureusement, de ce fait, notre environnement n'est pas en mesure de répondre de manière automatique à tout ce qui touche à des aspects proches de la présentation. Par exemple, la définition des paramètres du programme, bien qu'elle soit systématiquement identique quelle que soit l'implémentation, n'est pas mécanisée et doit réalisée par le concepteur. Pour compenser cette carence, nous avons mis au point un certain nombre de règles de développement définissant tout ce qui se rapporte à l'enregistrement d'un scénario. Nous expliquons dans cette partie quelles sont ces règles et comment les mettre en place dans une application basée sur H[4].

3.4.3.1 Règles de construction

D'un point de vue général les actions doivent être enregistrées au niveau de l'ANF, comme c'est le cas dans EBP. Cela permet d'obtenir une représentation de l'action indépendante de la Présentation et représentant une abstraction de celle du Noyau Fonctionnel.

L'ANF étant écrit par le concepteur, celui-ci doit intégrer, pour les actions qu'il désire voir enregistrer, les instructions nécessaires dans le programme de la métamorphose, ou plus précisément dans un des scénarii de ce programme.

Le domaine dans lequel nous nous trouvons présente des particularités qui nous permettent d'effectuer quelques simplifications vis-à-vis de l'enregistrement des actions.

D'une part, les objets étudiés sont des objets naturels structurés, c'est-à-dire qu'ils sont composés de sous-objets. On peut donc, à partir de l'objet initial accéder aux éléments qui le composent. Inversement, on peut retrouver un élément de « plus haut niveau » à partir

d'un de ses sous-composants. En terme de programmation, cela revient à faciliter considérable le travail du concepteur dans le passage des paramètres. En effet, une action s'effectuant sur un composant de la structure n'a pas besoin de comporter d'autre paramètre que l'élément concerné par l'action. Par exemple, dans le cadre de la croissance d'un arbre filaire, si l'on veut modifier la longueur d'un entrenœud donné, on retrouvera l'axe et éventuellement l'arbre auxquels il appartient grâce aux liens qui unissent les différents éléments de la structure. Le seul paramètre « entrenœud » suffit donc à la réalisation de l'action.

D'autre part, les composants d'un objet naturel structuré peuvent toujours être orientés. Ainsi, par exemple, les entrenœuds de l'axe d'un arbre possèdent un « numéro » à l'intérieur de cet axe. De ce fait, comme le montrent les exemples étudiés au chapitre 1 et présentés en annexe, on désigne toujours un élément de la structure par une méthode : on parle par exemple de l'avant-dernier entrenœud de l'axe principal. Cela a pour conséquence de simplifier singulièrement le problème de nomination des objets. Ceux-ci sont toujours assurés d'avoir le même nom puisqu'ils sont désignés, à l'intérieur du programme, relativement aux autres. On peut donc s'abstenir de la gestion de contexte dynamique présentée notamment dans EBP. Rappelons que celle-ci permet, en gérant les noms des objets créés de manière locale aux blocs et sous-blocs d'instruction d'un programme, de garantir qu'un objet aura toujours le même nom d'une exécution à l'autre.

L'enregistrement « type » d'une action se présente donc de la manière suivante. Le concepteur doit créer le questionnaire correspondant à l'action afin de l'intégrer dans le contrôleur de dialogue de son application. Ensuite, il doit écrire la méthode appelée par ce questionnaire lorsque celui-ci a reçu tous les jetons nécessaires à son exécution. Cette méthode se charge de transformer les jetons du contrôleur de dialogue en objets du Noyau Fonctionnel ainsi que d'appeler la primitive correspondante afin de rendre effective la réalisation de l'action. En plus de ce « transfert » classique, le concepteur doit décrire l'enregistrement de l'action. Il doit transformer les jetons reçus en jetons du Contrôleur de Scénario : un jeton commande correspondant à un questionnaire du contrôleur de scénario puis un jeton paramètre pour chaque objet intervenant dans l'action. Cette suite de jetons doit ensuite être ajoutée au scénario de l'objet sur lequel porte l'action. Il est évident que chaque action devant être enregistrée doit être représentée au sein du Contrôleur de Scénario par un questionnaire et que doit lui correspondre une action de l'Adaptateur de Scénario, dont le rôle sera de faire le transfert entre le scénario et le Noyau Fonctionnel.

Nous verrons par la suite les particularités liées à cet ajout et notamment comment choisir le scénario auquel l'action doit être ajoutée et comment sont gérés les jetons représentant un paramètre formel.

3.4.3.1.1 Cadre général d'exécution

Le cadre général de l'exécution d'un programme représentant la métamorphose d'un objet naturel structuré se présente sous la forme suivante :

```
Programme Métamorphose
        -- Créer l'objet initial
        O ← CréerObjetInitial (ListeDeParametres)
        Pour chaque top d'horloge Faire
                -- Exécuter les scénarii des sous-objets de O en le parcourant
                ParcourirEtExécuter (O)
        FinPour
FinProgramme
```

L'exécution du programme de croissance comprend donc, en plus du parcours de l'objet, la création initiale de l'objet. La primitive doit être enregistrée au moment de son exécution. Cela implique la présence d'une méthode permettant à l'utilisateur de rejouer la croissance depuis le début et qui soit capable de lire un scénario donné (le premier de la liste des scénarii) pour initialiser la métamorphose. Une autre primitive doit quant à elle permettre de faire avancer l'horloge et d'effectuer un nombre donné de tours de boucle. Il s'agit là d'une répétition implicite qui n'a pas à être commandée par l'utilisateur puisqu'elle est systématique et commune à toutes les métamorphoses.

Le système doit donc comporter :

- Une action permettant de créer l'objet initial. Lorsque celle-ci est activée par l'utilisateur, elle est enregistrée dans le premier scénario de la liste des scénarii.
- Une action permettant de rejouer la métamorphose depuis l'étape initiale. Celle-ci doit remettre à jour l'affichage, demander la valeur des paramètres du programme (nous verrons plus loin comment) et exécuter le premier scénario de la liste des scénarii.
- Une action permettant de faire avancer la métamorphose d'un certain nombre d'étape. Celle-ci doit donc simplement incrémenter l'horloge, parcourir la structure de l'objet principal et exécuter successivement les scénarii des éléments qui le composent.

La liste des paramètres liés à ce premier scénario correspond aux paramètres du programme global. Les scénarii attachés aux différents sous-objets de la structure peuvent se référer à lui pour connaître la valeur d'un paramètre donné.

3.4.3.1.2 Gestion des paramètres formels

La gestion des paramètres formels du programme représentant la métamorphose comporte trois étapes : la création des paramètres par l'utilisateur, l'utilisation de ces paramètres dans le cadre de la construction de l'exemple et enfin leur instanciation au moment où l'utilisateur décide de ré-exécuter la métamorphose.

La manipulation de paramètres formels, dont on veut pouvoir changer la valeur lors de la ré-exécution du programme, implique la présence d'un nouveau type de jeton, non prévu dans l'implémentation initiale du contrôleur de dialogue de H[4]. Exactement comme nous l'avons vu pour le Contrôleur de Scénario, ceux-ci doivent transporter, en plus d'une valeur représentant un objet du Noyau Fonctionnel, un nom qui les identifie. En terme de

programmation, il s'agit d'un jeton spécialisant, au même titre que les jetons « paramètres » classiques d'une classe jeton de plus haut niveau. Ceci autorise donc leur utilisation dans les mêmes conditions que les jetons classiques.

Création

L'utilisateur doit pouvoir à tout moment ajouter un paramètre formel au programme. Une commande doit donc lui permettre de créer un paramètre en spécifiant son nom et sa valeur. Celle-ci est indispensable : lors de la suite de la construction de l'exemple de métamorphose, l'utilisateur doit pouvoir faire appel aux paramètres en faisant référence à leur nom. Mais, pour que l'exécution de l'exemple continue, il faut que le paramètre possède une valeur. C'est elle qui est utilisée dans l'exécution courante mais c'est son identifiant qui est enregistré dans le programme.

Le contrôleur de dialogue doit posséder un interacteur permettant, à partir d'une chaîne de caractère et d'une valeur, de créer un paramètre formel et de l'ajouter à la liste des paramètres du premier scénario de la liste des scénarii.

Utilisation

L'interface doit disposer d'un objet de présentation permettant de visualiser la liste des paramètres formels du programme. Cet objet doit permettre à l'utilisateur de sélectionner un paramètre au moment où il intervient dans une action : la valeur courante du paramètre est alors utilisée pour que l'exécution de la métamorphose se poursuive mais c'est le nom formel qui est enregistré dans le scénario correspondant. Ainsi, lors de la ré-exécution du scénario, ce paramètre pourra prendre une valeur différente.

L'objet de présentation affichant la liste des paramètres doit donc, dès que l'utilisateur sélectionne une entrée, transmettre le jeton « paramètre formel » correspondant au moniteur du contrôleur de dialogue. Ce jeton sera utilisé par l'action en cours de réalisation (donc par le questionnaire en attente de ce type de jeton) et enregistré comme paramètre dans le scénario correspondant.

Instanciation

L'action permettant la ré-exécution totale de la métamorphose, à partir de l'étape initiale, doit demander l'instanciation de tous les paramètres présents dans la liste de paramètres du premier scénario de la liste de scénarii. La présentation doit donc disposer d'une méthode permettant de récupérer les valeurs correspondantes, par exemple avec une boîte de dialogue.

Au niveau du contrôleur de dialogue, cela se représente par un interacteur acceptant une chaîne de caractère et une valeur, et dont les questionnaires sont liés à une méthode permettant de modifier la valeur courante d'un paramètre à partir de son identifiant. La Présentation doit elle être en mesure de fournir au Moniteur des couples de jetons représentant l'identifiant du paramètre et sa nouvelle valeur.

3.4.3.1.3 Actions de base

A partir de ce que nous avons vu, nous pouvons préciser la méthode d'enregistrement d'une action de base. Lors de l'activation du questionnaire correspondant, la méthode de l'ANF se voit transmettre des jetons du contrôleur de dialogue. Ceux-ci sont de trois formes : commandes, paramètres ou paramètres formels.

Le jeton « commande », systématiquement présent, permet d'identifier la commande activée et donc de faire le lien avec la méthode du Noyau Fonctionnel. Si l'action doit être enregistrée dans le scénario, le concepteur enregistre cette commande.

Les jetons suivants sont ou bien des paramètres, ou bien des paramètres formels. Dans le premier cas, le concepteur doit simplement transformer le jeton en objet du noyau fonctionnel et enregistrer la valeur dans le scénario. Dans le second cas, il faut faire appel à la liste des paramètres du premier scénario de la liste des scénarii. On recherche la valeur de ce paramètre par rapport à son identifiant et on c'est elle que l'on transforme en objet du Noyau Fonctionnel. C'est par contre sous sa forme « formelle » qu'il est enregistré dans le scénario.

Un autre point important concerne l'identification du scénario. En effet, depuis le début de cette partie nous considérons un certain scénario, dans lequel les actions sont enregistrées. Mais, étant donné que le programme est constitué d'autant de scénarii que d'éléments « supports de scénario », il faut, lors de l'activation d'une action, être capable de déterminer à quel scénario on s'intéresse. On se donne donc la règle suivante : toute action sur un objet entraîne l'enregistrement de cette action au niveau du scénario de l'objet en question. S'il s'agit d'un objet n'étant pas associé directement à un scénario, l'action est enregistrée au niveau du premier objet englobant en supportant un (ce qui est possible en vertu des liens unissant les différents éléments de la structure d'un objet naturel structuré).

3.4.3.1.4 Création d'un support de scénario

La création de nouveaux sous-objets supportant un scénario implique la création du scénario, d'une part, et le lien entre le scénario de l'objet qui crée ce sous-objet et le scénario nouvellement créé, d'autre part. Dans l'exemple de la croissance d'un arbre filaire, ce sont les axes qui sont liés aux scénarii. Lorsque le scénario de croissance comporte l'insertion d'un nouvel axe, il faut être capable :

- Lors de l'enregistrement, de créer un nouveau scénario et de l'associer à l'axe inséré ;
- Lors de la phase de ré-exécution, de retrouver, lors de l'insertion de cet axe, le scénario gérant sa métamorphose.

Pour cette raison, l'action de création d'un élément lié à un scénario S doit comporter, en paramètre, l'identificateur de S.

3.4.3.1.5 Structures de contrôle

La mise en place de structures de contrôle nécessite l'introduction de deux composants : un premier permettant à l'utilisateur de décrire le prédicat de condition gérant la boucle ou la condition, un second lui permettant de spécifier la structure de contrôle utilisée.

Calculette grapho-numérique

D'un point de vue interactif, la description d'un prédicat peut être faite par l'intermédiaire d'une calculette grapho-numérique (telle que celles utilisées dans EBP ou TexAO). Il s'agit d'une calculatrice permettant d'effectuer des opérations logiques et mathématiques. La structure du Contrôleur de Dialogue de H[4] permet d'introduire très facilement un tel outil, et de l'intégrer de telle sorte que les opérandes puissent être aussi bien des constantes saisies directement avec la calculette, que des attributs d'objets ou des paramètres formels définis par l'utilisateur. En effet, il suffit pour cela de placer l'interacteur la représentant au sein du Moniteur de façon à ce qu'elle soit à même de recevoir les informations précitées d'interacteurs placés en dessous dans la hiérarchie.

Conditions et boucles

Les conditions et les boucles doivent également être gérées, au niveau du dialogue de l'application, par un interacteur. Celui-ci est activé par des commandes de la présentation permettant à l'utilisateur, par exemple par l'intermédiaire de menus, de spécifier les moments « clefs » de sa structure de contrôle (début d'une boucle, début d'une condition, fin des instructions du « sinon », etc.).

Deux problèmes principaux se posent pour l'enregistrement d'une structure de contrôle. Le premier concerne uniquement la structure conditionnelle et est liée à l'aspect graphique de sa description. En effet, la spécification d'une condition passe par plusieurs étapes :

- Initialisation de la phase d'enregistrement de la condition ;
- Saisie de la condition ;
- Description interactive des instructions à exécuter lorsque la condition est vraie ;
- Description interactive des instructions à exécuter lorsque la condition est fausse.

Or, interactivement, cela pose un problème : une fois décrites les instructions correspondantes à la première branche de la structure conditionnelle, il faut revenir en arrière pour se replacer dans le contexte de départ (celui d'avant la description de la première branche). Pour cela, il faut donc ou bien annuler toutes les actions exécutées, ou bien réexécuter le scénario jusqu'à ce qu'il revienne au début de la condition. La réponse la plus simple est de rejouer le scénario. Nous proposons de mettre en place une solution qui permet à l'utilisateur d'enregistrer les instructions de la première branche, et de ne décrire le comportement de la seconde branche que lors d'une exécution ultérieure, pour laquelle la condition s'avérera fausse.

Pour ça, l'utilisateur doit d'abord signaler la fin de la description de la branche « Alors ». Le jeton ajouté à la suite du scénario n'est alors pas « Sinon » mais « SinonTemporaire ». Il suffit ensuite de modifier le comportement de l'exécution du scénario lors de l'interprétation d'une structure alternative, pour que celui-ci soit capable de reconnaître ce nouveau jeton, et ainsi d'autoriser le système à rentrer dans une nouvelle phase de description.

L'enregistrement interactif d'une structure de contrôle pose un second problème : il n'est en effet pas possible de connaître a priori le scénario dans lequel elle doit être mémorisée. Une solution consiste donc à demander explicitement à l'utilisateur de désigner le composant lié au scénario devant enregistrer les instructions désirées. Cela revient à modifier l'interacteur modélisant la structure de contrôle dans le contrôleur de dialogue de façon à ce qu'il accepte un objet « support de scénario ». L'objet sera ainsi passé à la méthode de l'ANF correspondant, dans laquelle le concepteur sera à même de récupérer le scénario associé.

La même solution doit être adoptée pour la description du prédicat, dont on veut bien évidemment conserver le schéma de construction.

3.4.3.1.6 Lois d'interpolation

Comme nous l'avons vu lors de l'étude du chapitre 1 consacrée aux méthodes à base topologique, la modification des attributs géométriques s'effectue le plus souvent par l'intermédiaire de lois d'interpolation, linéaires ou non.

Dans la majorité des cas, les modèles d'objets naturels structurés représentent les attributs géométriques par des fonctions d'évolution. La description de l'interpolation par l'utilisateur affecte uniquement le scénario dans la mesure où elle spécifie le comportement de la fonction d'évolution d'un attribut entre deux bornes précises. De ce fait, l'unique travail du concepteur consiste à déduire de la description donnée une boucle du type :

```
Si top > 5 Alors

        -- Si on a dépassé l'étape 5 alors la longueur des entrenoeuds de l'axe est
        -- gérée par la fonction Fi

        Axe.AffecterFonctionLongueur ( Fi )

FinSi
```

Si les attributs géométriques des objets modélisés ne sont pas gérés par des fonctions, le concepteur doit ajouter au modèle une liste de fonctions. Il peut alors générer, à partir de la description interactive des interpolations, le « morceau » de programme fonctionnel correspondant, de la forme :

```
Si top > 5 Alors

        -- Parcours de la liste des EN de l'Axe

        Parcourir tous les entrenoeuds EN de l'axe courant AXE

                -- Modifier la longueur de l'entrenoeud courant en utilisant le résultat
                -- de la fonction d'interpolation Fi à l'instant top

                AXE.EN.ModifierLongueur ( Fi(top) )
```

```
        FinParcours
FinSi
```

Au niveau du dialogue, la description des lois d'interpolation est modélisée par un interacteur, dont les questionnaires, spécifiques à un attribut particulier (longueur, angle, épaisseur, etc.) attendent deux bornes, correspondant par exemple à deux étapes de la croissance dans le cas d'une interpolation dans le temps. Interactivement, cela peut se traduire par un menu proposant les différentes interpolations possibles, et demandant à l'utilisateur la désignation des bornes.

3.4.3.1.7 Visualisation des modifications

Lorsque l'utilisateur utilise une fonction du système, celle-ci déclenche une action de l'ANF. Si cette action est « enregistrable », le concepteur en prévoit d'une part l'exécution, pour que l'utilisateur ait un retour immédiat à l'écran, mais également son enregistrement dans un des scénarii composant la métamorphose globale. Selon le type de modification effectuée (géométrique, topologique), la fréquence de répétition de l'action correspondante dans le scénario ou l'étape à laquelle elle a été exécutée, le retour d'information sera visible sur une partie de la métamorphose seulement. De ce fait, il peut être nécessaire de rejouer la croissance depuis l'étape initiale pour que les transformations soient visibles et cohérentes avec l'ensemble des étapes de la métamorphose affichées.

3.4.3.1.8 Visualisation et Correction de programme

Dans la partie consacrée aux techniques de programmation sur exemple, nous avons insisté sur la nécessité, pour un utilisateur d'être en mesure de visualiser et de corriger le programme qu'il décrit interactivement. Le problème posé est qu'une représentation textuelle ne convient pas, étant donné que l'utilisateur n'a d'autre projection mentale de celui-ci que celle qu'il a utilisée pour le construire. Dans le contexte qui nous intéresse, le programme est représenté par l'exécution de la métamorphose qui reste suffisante pour l'utilisateur en ce qui concerne les actions de modification de base. Ceci étant, lorsqu'un des scénarii de la métamorphose contient une condition ou une boucle, cette représentation n'est plus suffisante : on ne peut visualiser précisément, dans le même mode graphique que les actions de base, les prédicats ou les débuts et fin de conditions ou de boucles.

Ce problème de visualisation est d'autant plus gênant lorsque l'on s'intéresse à d'éventuelles corrections à apporter au programme. Corriger une action de base peut en effet être réalisé de manière assez simple, par exemple en donnant au système une fonction d'exécution « pas à pas » détaillée et en permettant à l'utilisateur d'effacer la dernière action effectuée (un système de UNDO similaire a déjà été implémenté dans EBP). Mais le problème est plus compliqué s'il s'agit de corriger le prédicat d'une structure de contrôle, car celui-ci n'a aucune représentation graphique.

Etant donné que le mode de description des structures de contrôle et des prédicats les conditionnant n'est pas le même que celui des actions de base, il faut une solution adaptée. Les conditions et les boucles sont introduites de manière explicite et nécessitent donc

un traitement spécial. Une idée consiste à implémenter un mode d'exécution particulier qui affiche explicitement à l'écran le fait que le programme est en train d'évaluer un prédicat, qui l'affiche (sous une forme textuelle) et qui laisse la possibilité à l'utilisateur de le redéfinir ou de supprimer l'intégralité du bloc concerné.

3.4.3.1.9 Choix du programmeur

En plus de tous ces conseils que l'on peut donner au concepteur, un certain nombre de choix doivent rester à sa discrétion. Doit-il déduire des informations de certaines actions de l'utilisateur, car le contexte de son application ne laisse planer aucune ambiguïté sur ce que celui-ci peut faire à un moment donné de l'exécution du programme ? Par exemple, dans le cas de la croissance d'un arbre filaire, il peut très bien décider de fixer la périodicité d'une action, en fonction de l'étape à laquelle elle a été entreprise : si l'utilisateur ajoute un entrenœud au sommet d'un axe à l'étape 2, alors il faut répéter cette action sur cet axe à toutes les étapes paires. Au contraire, le concepteur veut-il que l'utilisateur possède un contrôle total ? Dans ce cas, il est de son recourt et de son choix de l'obliger à spécifier explicitement la périodicité des actions exécutées.

3.4.3.2 Conclusion

Parce que nous voulions que le concepteur de l'application puisse utiliser l'interpréteur de langage sur une modélisation préexistante sans qu'il n'ait même à en changer l'éventuelle interface, l'environnement que nous avons développé ne prend pas en compte les aspects liés à la Présentation. De ce fait, toutes les procédures d'enregistrement du programme que l'on pourrait facilement automatiser, se retrouvent cantonnées à l'état de conseils au concepteur.

Il s'agit d'une carence qui doit pouvoir être compensée de plusieurs façons possibles. Une première solution consiste par exemple à créer une partie de l'interface (celle dont on a besoin pour systématiser certaines actions d'enregistrement), et à laisser simplement libre la partie visualisation du modèle (en considérant par exemple que celle-ci s'effectue en OpenGl et en prévoyant donc en conséquence une fenêtre de visualisation autorisant l'affichage de primitives OpenGl). Une autre solution consiste à « outiller » encore notre environnement afin de permettre une mise en place plus interactive des règles que nous venons de voir.

3.5 Exemple d'application : un modeleur de croissance d'arbre en 1D½

Pour illustrer notre approche, nous avons développé une application permettant de modéliser et de manipuler des arbres filaires, en 1D½. Nous avons ensuite fait correspondre ce modèle au modèle générique de notre environnement, puis nous avons implémenté les composants permettant l'enregistrement des actions et la génération d'un programme de métamorphose (Figure 77).

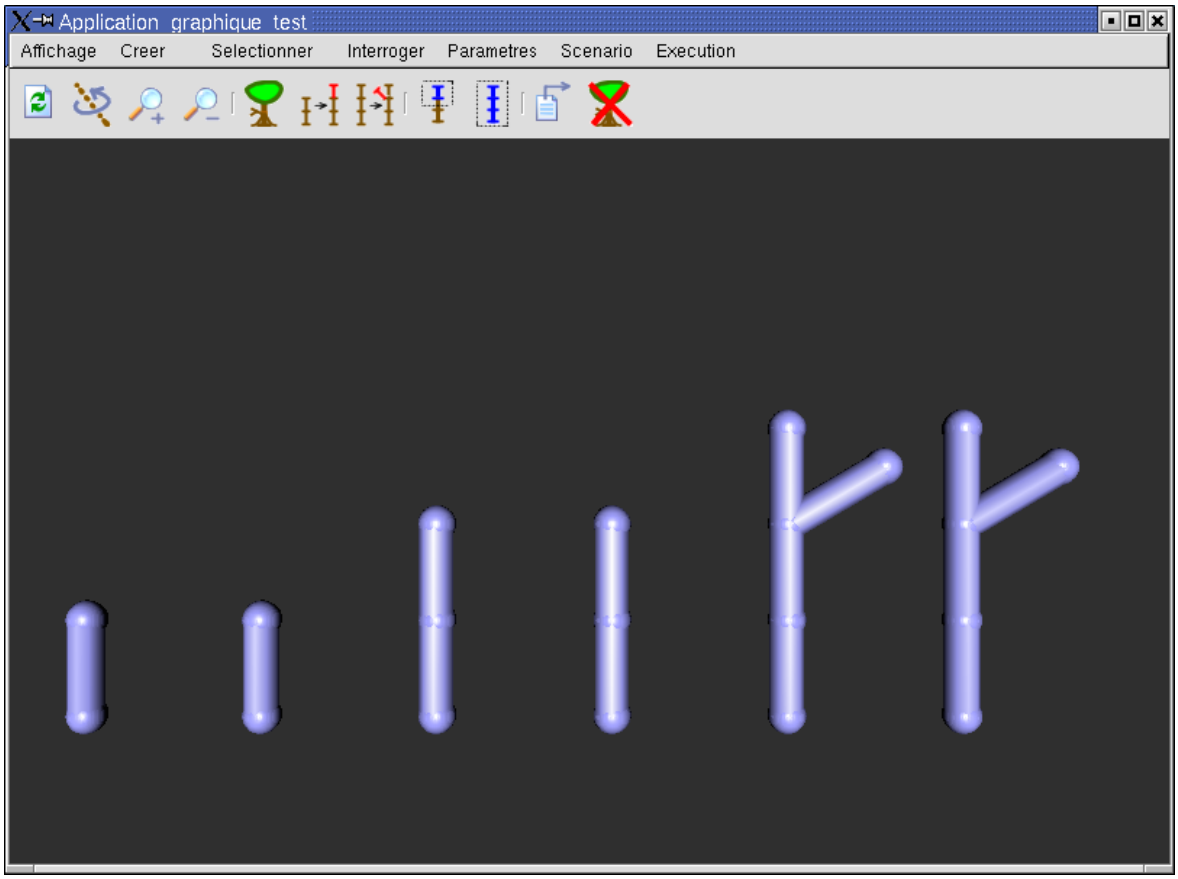


Figure 77 : Une application de simulation d'arbre filaire (plongement 3D) réalisée à partir de notre environnement

3.5.1 *Structures de données*

Le noyau fonctionnel de l'application est basé sur un modèle du type de celui de l'AMAP. Un arbre est ainsi décomposé en axes de différents ordres, eux-mêmes étant composés d'une succession d'entrenœuds.

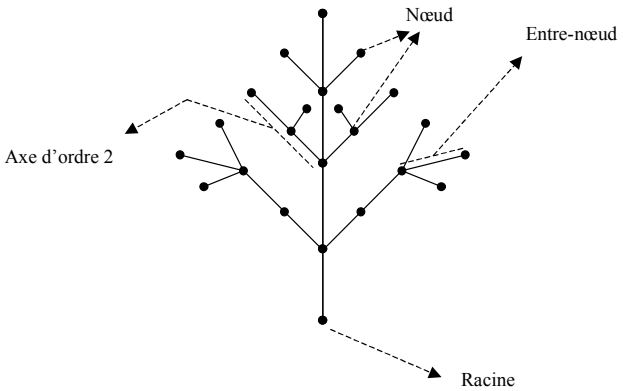


Figure 78 : Composants d'un arbre filaire, en 1D½

La Figure 79 montre le schéma UML des différents composants de l'application. Les structures du modèle générique qui sont ici instanciés ne sont pas détaillées, mais le lec-

teur peut se reporter au schéma de la section précédente pour se les remémorer (Figure 74).

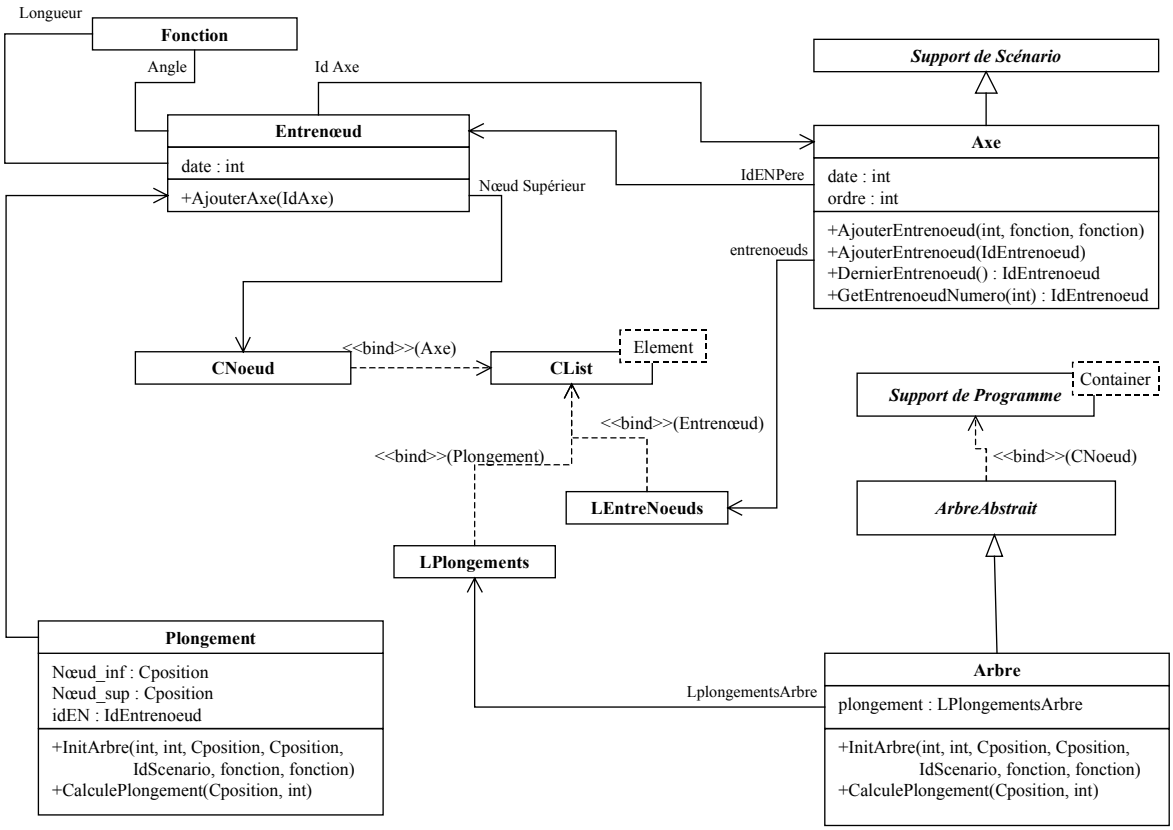


Figure 79 : Schéma UML du Noyau Fonctionnel

La structure générale est donc la suivante : un arbre est composés d'axes, caractérisés par un ordre. Les axes sont constitués d'entrenœuds lesquels possèdent eux-mêmes des liens vers de nouveaux axes. D'autre part, chaque entrenœud est lié à une structure de plongement qui permet de le représenter à l'écran.

3.5.2 Mapping avec le modèle générique

Pour réaliser la correspondance entre notre Noyau Fonctionnel et le modèle générique, il nous faut identifier, parmi les éléments de la structure de données, ceux qui :

- supportent le programme. Il s'agit d'un composant unique à partir duquel on peut initier le parcours entier de la structure, et donc exécuter le programme global. Dans notre cas, cet élément de départ est l'arbre.
- supportent un scénario. Ces composants, qui peuvent être de différentes natures, possèdent un comportement propre dont on veut décrire l'évolution. Ils seront liés à la structure de scénario. Le parcours de la structure globale doit permettre d'accéder à chacun d'entre eux de manière à en exécuter les différents comportements. Dans notre exemple, seul l'axe correspond à cette définition.

D'un point de vue implémentation, ces correspondances s'effectuent de la manière suivante.

Pour le support de programme, il faut dans un premier temps instancier la classe « Support de Programme » avec la structure de base. Ici, il s'agit d'une liste d'axes, la classe « CNoeud »). Cette instanciation nous donne une classe abstraite, en l'occurrence, la classe « ArbreAbstrait ». La dernière étape consiste simplement à spécifier le fait que notre classe « Arbre » de départ dérive de la classe « ArbreAbstrait » (Figure 80).

```
typedef SupportDeProgramme <CNoeud> ArbreAbstrait;

class CArbre : public ArbreAbstrait

        ////////////////////////////////////////////////////////////////////////
        // Propriétés
        // La classe CArbre définit un arbre de manière topologique et géométrique
        // Elle possède deux attributs : un noeud représentant la liste des axes
        // issus de la racine et une liste de plongement, contenant toutes les
        // infos géométriques des entrenoeuds de l'arbre.
        ////////////////////////////////////////////////////////////////////////
        {

        private:

        …
```

Figure 80 : Définition de la classe "Arbre"

La conséquence directe de cette dérivation est la nécessité de définir au niveau de la classe « Arbre » les méthodes déclarées dans « SupportDeProgramme » (Figure 81).

```
void CArbre::InitParcours (CNoeud *idCont) {
        …
};

bool CArbre::ParcoursEnCours () {
        …
};

IdSupportDeScenario CArbre::SupportDeScenarioSuivant () {
        …
};

void CArbre ::ExecuterScenario (ListParameter parametres) {
        …
};
```

Figure 81 : Définition des méthodes de "SupportDeScenario" dans "Arbre"

La seconde étape de la correspondance entre notre Noyau Fonctionnel et le modèle générique consiste à déclarer la classe « Axe » comme classe dérivée de « SupportDeScénario » (Figure 82).

```
class CAxe : public SupportDeScenario
{
        private :
        …
```

Figure 82 : Dérivation de la classe "Axe"

3.5.3 Description de l'environnement

L'application que nous avons développée permet donc à un utilisateur de décrire la métamorphose d'un arbre filaire, étape par étape. L'application a été développée en C++ et

l'interface en GTK. Le plongement graphique de l'arbre s'effectue dans un canevas permettant l'affichage de primitives OpenGl.

Dans cette application nous avons fait un certain nombre de choix d'implémentation. Tout d'abord, les modifications apportées au modèle sont immédiatement enregistrées. D'autre part, une modification ayant lieu à une étape donnée du développement fixe sa répétition (une action intervenant à l'étape 3, sera répétée à toutes les étapes multiples de 3).

L'utilisateur dispose d'un nombre limité d'actions de base :

- Création d'un arbre ;
- Ajout d'un entrenœud au sommet d'un axe ;
- Insertion d'un axe au sommet d'un entrenœud ;
- Modification géométrique des angles et longueurs d'un entrenœud.

Il a également la possibilité de définir à tout moment des paramètres qu'il peut utiliser ensuite dans la description du développement. D'autre part, il peut spécifier des structures conditionnelles.

Un certain nombre de fonctions permettent également d'accéder aux valeurs de certains attributs : taille d'un axe en terme d'entrenœuds, longueur d'un entrenœud, angle d'un entrenœud, âge d'un axe ou d'un entrenœud, etc. Celles-ci peuvent être utilisée notamment dans l'expression d'un prédicat contrôlant une condition.

Au niveau de la description des modifications géométriques, nous avons fait le choix d'utiliser des lois mathématiques, qui s'inscrivent en dehors des scénarii eux-mêmes. L'évolution de ces attributs géométriques peut être spécifiée grâce à des interpolations. L'utilisateur peut en effet modifier la longueur ou l'angle d'un entrenœud unique, puis demander au système de calculer l'interpolation linéaire correspondant à l'évolution de cet attribut entre deux étapes de la croissance, ou entre deux « étages » (positions dans l'axe) d'un axe à une étape donnée.

L'outil que nous avons développé à titre expérimental n'est qu'une maquette qui n'implémente pas la totalité des concepts étudiés précédemment. La description explicite des boucles ou la modification interactive des structures de contrôle sont, par exemple, encore en cours d'implantation.

Cette constatation nous amène à dire que, malgré toute l'aide que peut apporter l'environnement que nous avons développé, un grand travail reste encore à entreprendre pour améliorer son utilisabilité. La conclusion de ce manuscrit nous amènera à nous pencher de manière plus précise sur les multiples possibilités de développement futur permettant de résoudre ces problèmes.

4 Conclusion

Nous avons vu dans cette partie que l'architecture H4, malgré l'aide qu'elle peut apporter au concepteur d'une AGICT, ne répond pas à toutes les attentes possibles d'une applica-

tion interactive et n'offre pas tout le support souhaitable au développement de son contrôleur de dialogue.

En effet, le dialogue structuré est de nature complètement antagoniste à celle de la Manipulation Directe : le premier correspond à un langage préfixé et l'autre à un langage postfixé. Ceci étant, en modifiant le rôle du moniteur du contrôleur de dialogue et en spécifiant des interacteurs spécifiques, nous avons montré qu'il était possible de faire cohabiter ces deux modes de dialogue au sein de H4.

Un second problème concerne le développement même du contrôleur de dialogue de l'architecture. Celui-ci nécessite l'implémentation de plusieurs interacteurs et l'incorporation de nombreux questionnaires, ce qui revêt rapidement un caractère répétitif et surtout source d'erreurs. Pour cette raison, nous avons mis en place un outil graphique et interactif permettant à un concepteur de générer de manière automatique une partie du code du contrôleur de dialogue. Cet éditeur s'avère en outre capable de vérifier un certain nombre de propriétés du dialogue et prévient de la sorte une partie des erreurs de conception.

Une fois résolus ces problèmes, nous nous sommes penchés sur l'environnement d'aide à la création d'applications de métamorphose proprement dit. Celui-ci est constitué de plusieurs éléments : un modèle générique, reposant sur les propriétés communes des méthodes basées sur la topologie, un interpréteur de scénario, s'appuyant sur les caractéristiques de ce modèle, et un moteur d'enregistrement, qui se restreint, pour des raisons de généricité, à un certain nombre de règles et de conseils de développement.

Le modèle générique part du principe que les modèles utilisés dans les méthodes à base topologique sont structurés en sous-objets, dont certains sont liés à un comportement qui leur est propre. Il est, de ce fait, décomposé en deux modules : le premier permettant de parcourir la structure dans son intégralité, le second identifiant les éléments supportant une partie du programme global. A partir d'un modèle reposant sur ces principes généraux, un concepteur peut donc greffer son propre Noyau Fonctionnel sur ce modèle générique pour bénéficier des services de l'interpréteur de scénario.

Celui-ci est basé sur le fonctionnement du Contrôleur de Dialogue de H4. Nous avons montré comment il était possible de faire en sorte qu'il interprète des instructions séquentielles mais également des structures de contrôle (conditions et boucles). Les scénarii qu'il est ainsi capable de reconnaître sont indépendants du modèle apporté par le concepteur.

Pour construire le moteur d'enregistrement, il n'existe pas d'outils interactifs ou de modules prédéfinis. En voulant faire en sorte que le concepteur soit libre d'implémenter son application avec une présentation qui lui soit propre, nous avons restreint les possibilités d'automatisation. Pour le développer, il faut donc s'appuyer sur un certain nombre de principes propres à la Programmation sur Exemple mais dont nous avons expliqué les conséquences à l'intérieur du contexte qui nous intéresse.

Enfin, nous avons présenté une maquette développée selon les principes édictés dans cette partie. Celle-ci repose sur un Noyau Fonctionnel permettant de manipuler des arbres

filaires. Grâce à l'environnement que nous avons mis en place, celle-ci est en mesure d'être utilisée pour créer des métamorphoses de manière interactive.

Les limites présentées par le moteur d'enregistrement, qui se retrouvent dans l'implémentation de notre outil, sont discutées dans la conclusion finale de ce manuscrit.

Conclusion générale

Que ce soit à des fins commerciales (industrie du cinéma, publicité), culturelles (art contemporain) ou éducatives (étude de la collision de deux solides) la création d'animations est à l'origine d'un nombre important de travaux de recherche. Les méthodes qui en découlent sont aussi variées que nombreuses : prenant en compte des connaissances biologiques pour la simulation de phénomènes naturels, de lois physiques pour étudier la déformation de matériaux ou l'influence de forces extérieures pour articuler un squelette rigide.

La plupart de ces méthodes trouvent leur application dans des logiciels plus ou moins grand public, utilisant des voies interactives pour leur description. Ainsi, 3D Studio™, un des plus célèbres modeleur 3D destiné à des utilisateurs « finaux », permet il par exemple de mettre en pratique des techniques aussi complexes que la cinématique inverse ou l'interpolation par images clefs.

Toutes ces méthodes ne proposent cependant que d'effectuer des modifications géométriques sur les objets manipulés. Or, le domaine de la métamorphose, consistant à décrire l'évolution d'objets naturels structurés dans le temps, repose lui sur des modifications topologiques du modèle. Ces transformations particulières posent un problème encore non résolu, dans le sens où il n'existe aucune autre méthode de description que celle consistant à utiliser un langage de programmation. En effet, une étude précise de celles-ci montre que le caractère de leur contrôle est purement algorithmique, et qu'il gère de ce fait des concepts aussi complexes que la notion de variables, de paramètres ou de structures de contrôle.

Pour cette raison, nous nous sommes intéressé à la possibilité de créer un environnement d'aide à la programmation de telles simulations.

Dans ce mémoire, nous avons commencé par étudier les différentes techniques d'animation proposées par la littérature du domaine. Nous avons ainsi pu constater que la majorité d'entre elles ne s'intéressaient qu'aux modifications géométriques des objets dont elles modélisent l'évolution. D'autre part, ces principes trouvent leur application dans des systèmes « grand public » dans lesquels les concepteurs rivalisent d'ingéniosité pour permettre leur utilisation de la manière la plus conviviale possible. À l'opposé, seules les méthodes basées sur la topologie s'intéressent à la possibilité d'effectuer des modifications structurelles. Elles s'appuient sur une décomposition des objets en un ensemble

de sous-objets, chacun pouvant être porteur d'un comportement propre, correspondant à une partie de l'évolution globale, appelée métamorphose.

La contrepartie de cette puissance apparente est la difficulté éprouvée pour contrôler une telle animation. En nous fondant sur une étude de plusieurs méthodes basées sur la topologie, nous avons mis en exergue le caractère intrinsèquement algorithmique de ces approches. Ce fait avéré nous a permis de comprendre la difficulté de mettre en pratique la description de ces transformations de manière interactive et, de ce fait, l'absence quasi-totale de systèmes conviviaux de modélisation de métamorphoses. La pratique usuelle consiste en effet à prendre son langage de programmation préféré et à coder, à la main, toutes les instructions nécessaires. Dans un domaine aussi visuel que celui de l'animation, une telle méthode n'est évidemment pas satisfaisante.

Pour terminer cette analyse, nous avons enfin réalisé une étude de cas, laquelle nous a permis de faire ressortir les concepts informatiques nécessaires à la description de simulations de phénomènes naturels basées sur la topologie.

Dans le but de mettre la programmation, puisque c'est comme ça qu'il convient de l'appeler, de métamorphose à la portée d'un plus grand nombre d'utilisateurs et, en tout cas, de faire en sorte qu'elle devienne moins pénible aux experts du domaine, nous avons cherché les outils de conceptions qui permettrait de combler cet écueil.

Nous avons donc d'abord cherché à replacer les applications de création de métamorphoses dans un cadre connu, et nous avons montré qu'il s'agissait d'une AGICT : une Application Graphique de Conception Technique, laquelle devait, au même titre qu'un système de CAO, permettre de définir des besoins précis autant que complexes. Dans ce contexte, le dialogue de l'application se devait de suivre le principe de Norman, selon lequel un utilisateur doit avoir la possibilité de décomposer une tâche de haut niveau en un ensemble plus accessible de buts / sous-buts.

Une telle démarche, que recouvre le terme de Dialogue Structuré, n'est pas facile à mettre en place. Nous avons étudié différentes architectures logicielles pour savoir dans quelle mesure l'une d'entre elles pouvait aider un concepteur dans la réalisation de ce travail. Seule H[4], une architecture hybride, apporte un support au concepteur dans la modélisation d'un dialogue structuré. Le Contrôleur de Dialogue permet la répartition des tâches dans des modules logiciels, les Interacteurs, représentant différents niveaux d'abstraction. Ils sont indépendants entre eux et communiquent grâce à un module externe, le Moniteur. Ce fonctionnement assure qu'une tâche peut intervenir dans la construction d'une autre tâche sans que cela ait été prévu par le concepteur. D'autre part, ajouter une fonction à ce Contrôleur de Dialogue revient simplement à définir sa signature, le Questionnaire, en indiquant la méthode de l'ANF correspondante, et à l'insérer dans un interacteur.

A partir des conclusions de l'étude menée dans le chapitre 1, nous avons cherché à savoir s'il existait une solution au problème de la définition interactive d'algorithmes tels que ceux qui définissent une métamorphose. Nous nous sommes donc intéressé à une classification des systèmes de programmation interactive et nous avons ainsi pu caractériser les applications de programmation de métamorphoses. Cette définition nous a permis de

comprendre que seule la Programmation sur Exemple était en mesure d'apporter des solutions suffisamment puissantes pour permettre l'élaboration interactive de véritables programmes informatiques.

Nous nous sommes ensuite penchés sur les concepts fondamentaux de la Programmation sur Exemple en nous attardant sur les difficultés qu'elle pose (nomination des objets, distinction entre paramètres, variables, constantes, utilisation des structures de contrôle, visualisation et correction de programmes) et les solutions de la littérature.

Dans la dernière partie, nous avons commencé par étudier tour à tour les difficultés posées par H4 pour la réalisation d'un système interactif offrant des possibilités plus conviviales d'une part, et pour le développement même de son Contrôleur de Dialogue, d'autre part.

Le premier de ces deux problèmes consistait à intégrer dans le fonctionnement du Contrôleur de Dialogue de H4, une méthode permettant de l'ouvrir à des styles de dialogue plus conviviaux que l'unique dialogue structuré. En effet, si celui-ci permet à l'utilisateur d'exprimer les besoins précis nécessaires au contexte technique dans lequel il se trouve, certaines tâches, plus anodines, comme le simple zoom d'une scène, doivent pouvoir être réalisées dans un cadre moins rigide. Grâce à l'intégration d'interacteurs spécialisés et au renforcement du rôle du Moniteur, nous avons démontré la possibilité de faire cohabiter Manipulation Directe et Dialogue Structuré au sein du Contrôleur de Dialogue de H4.

Dans un deuxième temps, nous avons montré les difficultés d'implémentation du Contrôleur de Dialogue de H4. Celui-ci, lorsqu'il contient un nombre important d'interacteurs et de questionnaires, demande un code lourd et extrêmement répétitif, source d'erreurs non seulement syntaxiques, mais également organisationnelles au niveau des modules réalisés. Nous avons donc conçu un outil graphique et interactif capable de guider le concepteur d'une application reposant sur H4 dans la construction du Contrôleur de Dialogue. Cet éditeur génère une grande partie du code de celui-ci et opère également à des vérifications de base sur les propriétés du dialogue, évitant ainsi un grand nombre d'erreurs.

Enfin, ce manuscrit décrit l'environnement que nous avons réalisé. Plus qu'une simple application permettant de réaliser des métamorphoses, de les enregistrer et de les rejouer, nous avons mis en place un environnement permettant au concepteur d'une application de greffer son propre Noyau Fonctionnel sur un modèle générique, et de profiter ainsi des bénéfices d'un interpréteur de scénario. Ce cadre décrit également les propriétés que doit respecter le moteur d'enregistrement : il s'agit de concepts de programmation sur exemple rapportés à notre étude, mais ne présentant encore aucun outil interactif de mise en œuvre.

Le modèle générique se fonde sur des propriétés établies sur les modèles basés sur la topologie. Il reprend la structuration d'objets en sous-objets, et l'association de comportements à certains éléments de la structure. Il spécifie également l'exécution globale du pro-

gramme comme correspondant au parcours de l'objet et à l'application systématique des comportements rencontrés.

A ce modèle est associé un interpréteur de scénario. Celui-ci se base sur le fonctionnement du Contrôleur de Dialogue de H4, véritable interpréteur de langage. Cette adaptation nous permet notamment d'utiliser DTS Edit pour construire notre Contrôleur de Scénario. Les scénarii considérés sont, d'autre part, indépendants de la réalisation du Noyau Fonctionnel.

Enfin, nous associons à cet ensemble une description précise des éléments à prendre en compte pour permettre au concepteur de l'application de mettre en œuvre facilement l'enregistrement des métamorphoses décrites par l'utilisateur, sous la forme des scénarii fournis. Ce dernier composant, le moteur d'enregistrement, prend le parti de laisser au concepteur la plus grande liberté possible concernant la Présentation. Il est ainsi en mesure de choisir l'interface, la librairie d'affichage, etc.

La toute dernière partie démontre la validité de notre approche sur une maquette permettant de représenter, d'enregistrer et de rejouer des métamorphoses d'arbres filaires de manière interactive.

Notre étude se plaçait à la croisée de deux domaines : la modélisation d'animation et plus particulièrement de métamorphoses d'objets naturels structurés d'une part, et la Programmation sur Exemple d'autre part, mais notre cheminement nous a également amené à travailler sur les modèles d'architecture.

Nous sommes parvenu à démontrer la possibilité d'utiliser la Programmation sur Exemple dans le cadre d'un domaine jusque là totalement dénué d'interactivité. D'autre part, nous avons mis en œuvre une solution originale dans le cadre de la Programmation sur Exemple.

La difficulté posée par la multiplicité des différents domaines approchés ne nous a pas permis de mener jusqu'au bout toutes nos introspections. La richesse de ces univers nous ouvre néanmoins de nombreuses perspectives qui mériteraient d'être approfondies.

Au niveau des outils d'abord. DTS Edit doit être modifié pour permettre une modélisation facile de la manipulation directe. Les interacteurs spécialisés devraient y être prédéfinis, et des opérations adaptées à leur spécificité devraient y être insérées. D'autre part, on peut envisager compléter sa tâche en lui ajoutant des concepts de validation. Dans [Jambon, Girard, & Boisdron 1999], les auteurs montrent comment il est possible, avec H4, de vérifier « l'atteignabilité » et la complétude des interactions. Cette méthode pourrait être adjointe à l'éditeur. Enfin, il semble envisageable d'automatiser la création de certains jetons par introspection des classes du Noyau Fonctionnel.

Il reste également encore à envisager les multiples apports des similitudes entre le Contrôleur de Dialogue et ce que nous avons appelé le Contrôleur de Scénario. En effet, ils sont basés sur un fonctionnement identique, et implémentent de plus un grand nombre de questionnaires communs. Que la commande vienne d'un scénario ou d'un utilisateur

ne change rien pour le Moniteur. De plus, la même méthode du Noyau Fonctionnel est finalement appelée par l'une et par l'autre. La seule différence se trouve au niveau de l'ANF et de l'AdS. Dans le cas de l'ANF la méthode aura un rôle supplémentaire : généraliser et enregistrer l'action dans un scénario. Cette similitude nous permet de dire qu'il est possible de générer l'AdS à partir de l'ANF.

Ces similitudes nous permettent également de penser qu'il serait possible d'utiliser DTS Edit pour créer de manière interactive le Contrôleur de Scénario.

Dans le registre du moteur d'enregistrement, plusieurs ouvertures sont également envisageables. Comme nous l'avons vu, le fait d'avoir laissé la liberté à l'utilisateur d'utiliser la Présentation qu'il souhaitait nous a bridé dans l'automatisation de l'enregistrement des actions. Une conséquence de ce choix, comme nous l'avons constaté lors de la création de la maquette, est qu'il est difficile de mettre en œuvre rapidement les principes énoncés. Pour parvenir à un environnement beaucoup plus convivial au niveau de ce composant, plusieurs approches sont envisageables.

Une première solution consiste à outiller l'environnement, de manière à permettre la génération du code correspondant à l'enregistrement des actions.

En prenant le problème sous un angle différent, on peut imaginer fournir un cadre d'application plus contraignant mais aussi plus complet. Celui-ci gérerait la Présentation, à laquelle le concepteur devrait s'adapter, mais comporterait en contrepartie les mécanismes d'enregistrement des structures conditionnelles, des prédicats de contrôle ou de gestion des paramètres.

Dans un registre plus général, un travail intéressant consisterait à étudier la possibilité d'introduire des mécanismes de programmation par contrainte. Par exemple, il est envisageable de permettre à l'utilisateur de définir la forme globale d'un arbre et, à partir de ces informations, fournies de manière interactive, de générer le programme correspondant.

Une dernière perspective qu'ouvrent nos travaux concerne l'utilisation de cartes généralisées. Notre approche actuelle consiste à utiliser un modèle générique sur lequel le concepteur greffe le sien. Les cartes généralisées peuvent, elles, être utilisées pour modéliser tout type de métamorphoses ([Terraz 1994]). Elles présentent la particularité d'être définies à partir de quelques opérations de base seulement. On peut, grâce à ce mécanisme, définir tout type d'objet structuré ainsi que les opérations qui permettent sa transformation. Il semble donc raisonnable de penser qu'il est possible de mettre en place un environnement travaillant sur ce modèle et permettant la définition interactive d'objets et de primitives de manipulation basées sur celui-ci. Un moteur de Programmation sur Exemple tel que celui que nous avons décrit dans ce manuscrit pourrait alors être utilisé pour permettre l'enregistrement, la généralisation et le rejeu des opérations de transformations décrites.

Annexes

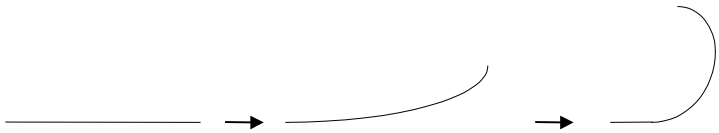
Stratégie globale sur les arbres

A chaque top d'horloge, on parcourt tous les axes. En fonction de leur ordre, on effectue un certain nombre d'opérations (qui peuvent demander de parcourir toutes les arêtes d'un axe par exemple).

Notation

Interpolation [va, va'] -> *[vb, vb']* : Signifie que l'on donne à une composante B variant entre vb et vb', la valeur correspondant à l'interpolation en fonction d'une composante A variant entre va et va'.

1 Exemple 1



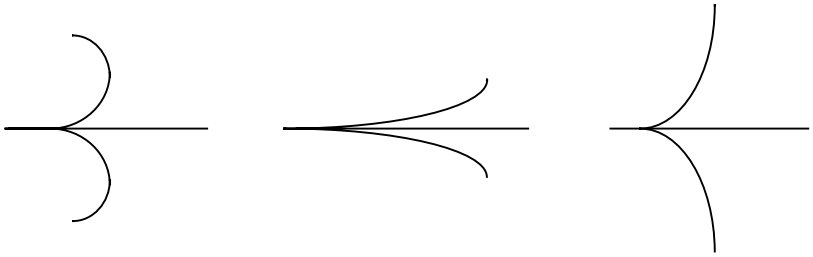
1.1 Description informelle

A partir d'une certaine date, l'angle des arêtes augmente en fonction de la distance à l'origine de l'axe et de l'âge de l'axe.

1.2 Pseudo algorithme

```
Pour chaque top d'horloge Faire

        -- On doit d'abord connaître l'angle de chaque arête en fonction de sa distance à l'origine de l'axe à sa
                -- date de maturité

        Interpolation [DistanceMin, DistanceMax] -> [Angle0, AngleMax] (à t = tmaturité)

        -- Ensuite, connaissant l'angle Max et Min de chaque arête au cours de son développement (Min donné au
        -- départ et Max donné par l'interpolation précédente), on calcule l'angle d'une arête à un moment
        -- quelconque de son processus de croissance : on utilise donc une interpolation en fonction de l'âge de
        -- l'arête (pour chaque arête, on connaît les angles correspondants à ses âges min et max).

        Interpolation [Age0, AgeMax] -> [Angle0, AngleMax]

FinPour
```

2 Exemple 2



2.1 Description informelle

- Plus la distance à l'origine de l'axe est grande, plus l'angle formé par l'arête est important et plus cet angle varie rapidement. Les angles varient entre deux valeurs opposées [-A, +A].
- L'angle de l'arête situé à une distance D de l'origine est plus important et varie plus rapidement que les autres. Les angles varient entre deux valeurs opposées [-A, +A].
- Plus la distance à l'origine de l'axe est grande, plus l'angle formé par l'arête est faible et plus cet angle varie lentement. Les angles varient entre deux valeurs opposées [-A, +A].

2.2 Pseudo algorithme

```
Pour tous les tops d'horloge Faire
        -- Pour chaque arête : Angle formé en fonction de la distance à l'origine de l'axe à t = Textremite

        Interpolation [DistanceMin, DistanceMax] -> [AngleDMin, AngleDMax] (à t = Textrémité2)

        -- Pour chaque arête : Angle formé en fonction de son âge (connaissant les angles min et max au cours de
        -- son évolution).

        Interpolation [AgeExtremite1, AgeExtremité2] -> [-AngleMax, AngleMax]

        -- Il faut prolonger ce résultat à tous les tops d'horloge. Comment faire ? Comme suit: tout ce qui est
        -- vrai à un instant t, l'est également à t modulo (AgeExtremite2 - AgeExtremite1) et à
        -- (-t) modulo (AgeExtremite2 - AgeExtremite1) :
        -- Interpolation [AgeExtremite1 modulo (NTO), AgeExtremite2 modulo (NTO)] -> [-AngleMax, AngleMax]
        -- Interpolation [-AgeExtremite2 modulo (NTO), -AgeExtremite1 modulo (NTO)] -> [-AngleMax, AngleMax]
        -- Où NTO est le nombre de tops d'horloge durant une oscillation = AgeExtremite2 - AgeExtremite1
FinPour
```

3 Photo 1

3.1 Description informelle

- Tronc : ajout d'arêtes jusqu'à insertion des axes d'ordre 2. Croissance des arêtes continue jusqu'à une certaine longueur (ou un certain âge). Insertion d'axes d'ordre 4 à l'extrémité de l'axe.
- Axes ordre 2 : ajout d'arêtes jusqu'à insertion des axes d'ordre 3. Croissance des arêtes continue jusqu'à une certaine longueur (ou un certain âge). Insertion d'axes d'ordre 4 à l'extrémité de l'axe.
- Axes ordre 3 : ajout d'arêtes jusqu'à insertion des axes d'ordre 4. Croissance des arêtes continue jusqu'à une certaine longueur (ou un certain âge).
- Axes ordre 4 : ajout d'arêtes avec un angle dépendant de la distance à l'origine de l'axe. Insertion d'axes d'ordre 5 à certains moments. Croissance des arêtes continue jusqu'à une certaine longueur (ou un certain âge).
- Axes ordre 5 : ajout d'arêtes avec un angle dépendant de la distance à l'origine de l'axe. Croissance des arêtes continue jusqu'à une certaine longueur (ou un certain âge).

3.2 Pseudo algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>
                        -- Ici, même quand l'axe a atteint son âge de maturité, certaines de ses arêtes peuvent
                        -- continuer à croître.
                        Pour toutes les arêtes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour
```

```
                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete
                        SinonSi Age(AX) = AgeMax Alors
                                Ajouter_n_Axes (Ordre => 2, Nbre => 3)
                                Ajouter_n_Axes (Ordre => 4, Nbre => 4)
                        FinSi

                2 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete
                        SinonSi Age(AX) = AgeMax Alors
                                Ajouter_n_Axes (Ordre => 3, Nbre => 3)
                                Ajouter_n_Axes (Ordre => 4, Nbre => 4)
                        FinSi

                3 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete
                        SinonSi Age(AX) = AgeMax Alors
                                Ajouter_n_Axe (Ordre => 4)
                                Ajouter_Axe (Ordre => 4)
                        FinSi

                4 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Longueur(AX) < LongMax) et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete(Angle => f(Distance(Origine(AR), Origine(Axe))
                                -- Où f est une fonction croissante
                                Ajouter_Axe(Ordre => 5)
                        FinSi

                5 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                Autres => Rien

                FinSelon
        Fin Pour
FinPour
```

4 Photo 2

4.1 Description informelle

- Tronc : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille). Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 2 à une certain fréquence.
- Axes d'ordre 2 : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille), selon un angle dépendant de la distance à l'origine de l'axe. Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 3 à une certaine fréquence. D'autre part, l'angle des arêtes semble dépendre de la taille de l'axe et de la distance à l'origine de l'axe. Plus les arêtes sont créées loin de l'origine de l'axe, plus l'angle est important. Plus l'axe est long, plus l'angle des arêtes situées à proximité du tronc baisse.
- Axes d'ordre 3 : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine longueur), selon un angle dépendant de la distance à l'origine de l'axe : plus on s'éloigne, plus l'angle de création se rapproche de 0. Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur).

4.2 Pseudo algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire
```

```
                1 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete
                                Ajouter_Axe (Ordre => 2)
                                -- Ou l'inverse ...
                        FinSi

                2 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0)) Alors
                                Ajouter_Arete (Angle => f(Distance(Origine(AR), Origine(AX))))
                                -- f croissante
                                Ajouter_Axe (Ordre => 3)
                        FinSi

                        -- Toutes les fleurs de l'arbre apparaissent vraisemblablement à la même date. Il en
                        -- apparaît un certain nombre à l'insertion de chaque axe.
                        Si (Date = Date_Floraison) Alors
                                Ajouter_Fleur (Noeud => Origine(AX), Nbre => Random)
                        FinSi

                3 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0)) Alors
                                Ajouter_Arete ( Angle => f(Distance (Origine(AR), Origine(AX))))
                                -- f décroissante
                        FinSi

                        -- Toutes les fleurs de l'arbre apparaissent vraisemblablement à la même date. Il en
                        -- apparaît un certain nombre à l'insertion de chaque axe.
                        Si (Date = Date_Floraison) Alors
                                Ajouter_Fleur (Noeud => Origine(AX), Nbre => Random)
                        FinSi

                Autres => Rien
                FinSelon
        FinPour
FinPour
```

5 Photo 3

5.1 Description informelle

- Tronc : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille). Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 2 à une certaine fréquence.
- Axes d'ordre 2 : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille), et selon un angle qui est modifié au cours du temps : plus l'arête est vieille et plus l'angle diminue. Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 3 à une certain fréquence. L'arête terminale (et seulement elle) est recouverte d'épines. Un axe perd des axes d'ordre 3 à partir d'un certain âge et selon la proximité du nœud de leur insertion par rapport à l'origine de l'axe.
- Axes d'ordre 3 : Même comportement que les axes d'ordre 2, avec une longueur plus limitée.

Remarque : Si l'on considère que les épines sont des arêtes normales appartenant à des axes d'ordre 4 par exemple, on a un problème pour dire que seules les p dernières arêtes de l'axe d'ordre 1 ont des axes incidents d'ordre 4 (alors qu'on pourrait se contenter de dire que la dernière arête de l'axe d'ordre 1 a des épines (ce qui correspond plus à la réalité je pense). Mais dans ce cas, on ajoute une entité à la structure : les épines).

5.2 Pseudo algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire
```

```
                1 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Supprimer_Epines (Arete_Terminale (AX))
                                Ajouter_Arete (AX)
                                Ajouter_Epines (Arete_Terminale (AX))
                                Ajouter_Axe (Ordre => 2)
                        FinSi

                2 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                                Interpolation [Age0, AgeMax] -> [Angle0, AngleN]
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Supprimer_Epines (Arete_Terminale (AX))
                                Ajouter_Arete
                                Ajouter_Epines (Arete_Terminale (AX))
                                Ajouter_Axe (Ordre => 3)
                        FinSi

                        Si (Long(AX) > LongMax) Alors
                                Pour toutes les aretes « AR » entre Origine(AX) et (Long(AX) - LongMax) Faire
                                        Supprimer_Arete (AR)
                                FinPour
                        FinSi

                3 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                                Interpolation [Age0, AgeMax] -> [Angle0, AngleN]
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Supprimer_Epines (Arete_Terminale (AX))
                                Ajouter_Arete
                                Ajouter_Epines (Arete_Terminale (AX))
                                Ajouter_Axe (Ordre => 3)
                        FinSi

                        Si (Long(AX) > LongMax) Alors
                                Pour toutes les aretes « AR » entre Origine(AX) et (Long(AX) - LongMax) Faire
                                        Supprimer_Arete (AR)
                                FinPour
                        FinSi

                Autres => Rien

                FinSelon
        FinPour
FinPour
```

6 Photo 4

6.1 Description informelle

- Tronc : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille). Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 2 et d'ordre 6 une fois l'ajout d'arête terminé.
- Axes d'ordre 2 : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille). Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 3 et d'ordre 6 une fois l'ajout d'arête terminé.
- Axes d'ordre 3 : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille). Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 4 et d'ordre 6 une fois l'ajout d'arête terminé.
- Axes d'ordre 4 : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille). Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 5 à une fréquence donnée. Modification des arêtes en fonction de leur position sur l'axe : l'arête terminale (et seule l'arête terminale) est recouverte d'épines.
- Axes d'ordre 5 : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille). Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Modification des arêtes en fonction de leur position sur l'axe : l'arête terminale (et seule l'arête terminale) est recouverte d'épines.

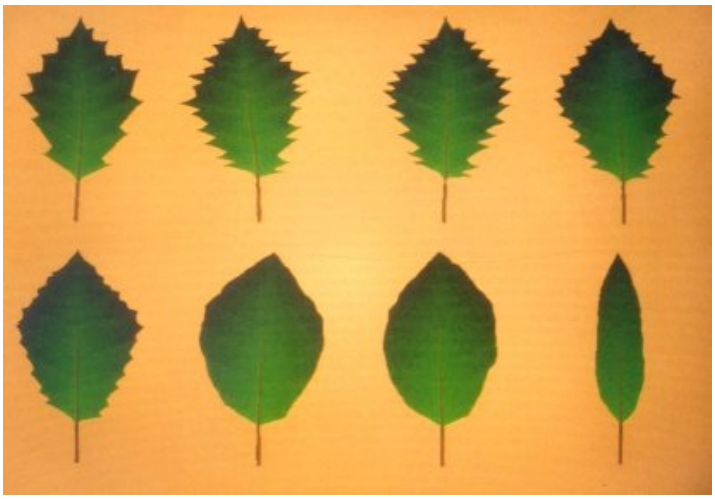
Remarque : La même concernant les épines que celle de la photo précédente.

6.2 Pseudo algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete
                        SinonSi Age(AX) = AgeMax Alors
                                Ajouter_n_Axes (Ordre => 2, Nbre => 2)
                                Ajouter_n_Axes (Ordre => 5, Nbre => Random)
                        FinSi

                2 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete
                        SinonSi Age(AX) = AgeMax Alors
                                Ajouter_n_Axes (Ordre => 3, Nbre => 2)
                                Ajouter_n_Axes (Ordre => 5, Nbre => Random)
                        FinSi

                3 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete
                        SinonSi Age(AX) = AgeMax Alors
                                Ajouter_n_Axes (Ordre => 4, Nbre => 2)
                                Ajouter_n_Axes (Ordre => 5, Nbre => Random)
                        FinSi

                4 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Supprimer_Epines (Arete_Terminale (AX))
                                Ajouter_Arete
                                Ajouter_Epines (Arete_Terminale (AX))
                                Ajouter_Axe (Ordre => 5)
                        FinSi

                5 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
```

```
                                    Sinon
                                        LongMax
                                    FinSi
                            FinPour

                            Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                    Supprimer_Epines (Arete_Terminale (AX))
                                    Ajouter_Arete
                                    Ajouter_Epines (Arete_Terminale (AX))
                            FinSi

                    Autres => Rien

                    FinSelon
            FinPour
FinPour
```

7 Photo 5



7.1 Description informelle

- Tronc : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille). Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 2 à une fréquence donnée.
- Axes d'ordre 2 : Ajout d'arêtes à une certaine fréquence jusqu'à ce que la taille (en terme de nombre d'arête) de l'axe soit égale à la distance entre l'origine de l'axe et la racine. Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'un axe d'ordre 3 à une date t à partir du ième nœud de l'axe (en partant de la fin). A partir d'une certaine date, suppression des arêtes terminales à une vitesse proportionnelle à la taille de l'axe.
- Axes d'ordre 3 : Ajout d'arêtes à une certaine fréquence jusqu'à ce que la taille soit égale à la distance entre l'origine de l'axe et l'extrémité de l'axe support. Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur).

Remarque : entre l'étape 3 et l'étape 6, on a création progressive d'une surface entre les axes d'ordre 2 et les axes d'ordre 3 qu'ils supportent.

Remarque : les axes d'ordre 3 ne sont supprimés que si l'arête qui les supporte est supprimée.

7.2 Pseudo algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) Alors
                                Si Date Modulo Frequence1 = 0 Alors
                                        Ajouter_Arete
                                FinSi
                                Si Date Modulo Frequence2 = 0 Alors
                                        Ajouter_Axe (Ordre => 2)
                                FinSi
                        FinSi

                2 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < Age1 Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) = AgeN1) Alors
                                Ajouter_Axe (Ordre => 3, Noeud => i)
                        FinSi
                        Si (Age(AX) <= AgeN2) Alors
                                Si (Taille(AX) < f(AX, Axe (Ordre => 1))) et (Date Modulo Frequence = 0) Alors
                                -- Avec f(AX, Axe(Ordre => 1) = Distance(Origine(AX),Origine(Axe(Ordre => 1)))
                                        Ajouter_Arete
                                FinSi
                        SinonSi (Age(AX) > AgeN2) et (Date Modulo f(Taille(AX))= 0) Alors
                                -- f est une fonction croissante. Plus la taille de l'axe est grande, plus la
                                -- fréquence de suppression des arêtes est élevée.
                                Supprimer_Arete (Arete_terminale(AX))
                        FinSi

                3 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < Age1 Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si Taille (AX) < Distance (Origine (AX), Extremite (Axe_Support(AX))) Alors
                                Ajouter_Arete
                        FinSi

                Autres => Rien

                FinSelon
        FinPour
FinPour
```

8 Photo 6

8.1 Description informelle

- Tronc : Ajout d'arêtes à une certaine fréquence jusqu'à une certaine date (ou jusqu'à une certaine taille). Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur). Insertion d'axes d'ordre 2 et d'un axe d'ordre 3 à la fin de la croissance de l'axe.
- Axes d'ordre 2 : Ajout d'arêtes à une fréquence F1 avec un angle nul et à une autre fréquence F2 avec un angle non nul jusqu'à une certaine date (ou jusqu'à une certaine taille). Insertion d'un axe d'ordre 3 à la fréquence F2. Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur).
- Axes d'ordre 3 : Ajout d'arêtes à une fréquence donnée jusqu'à une certaine date (ou une certaine taille). Insertion d'axes d'ordre 4 à une certaine fréquence. Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur).
- Axes d'ordre 4 : Ajout d'arêtes à une fréquence donnée jusqu'à une taille donnée (étage calculé) ne pouvant être supérieure à la distance entre l'origine de l'axe et l'origine de l'axe support. Croissance des arêtes continue jusqu'à un certain âge (ou une certaine longueur).

8.2 Pseudo algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete
                        SinonSi Age(AX) = AgeMax Alors
```

```
                                Ajouter_n_Axes (Ordre => 2, Nbre => 2)
                                Ajouter_Axe (Ordre => 3)
                        FinSi

                2 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) Alors
                                Si (Date Modulo Frequence1 = 0) Alors
                                        Ajouter_Arete (Angle => 0)
                                SinonSi (Date Modulo Frequence2 = 0) Alors
                                        Ajouter_Arete (Angle => Val)
                                        Ajouter_Axe (Ordre => 3)
                                FinSi
                        SinonSi (Age AX) = AgeMax) Alors
                                Ajouter_n_Axe (Ordre => 3, Nbre => 2)
                        FinSi

                3 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Age(AX) < AgeMax) Alors
                                Si (Date Modulo Frequence1 = 0) Alors
                                        Ajouter_Arete
                                SinonSi (Date Modulo Frequence2 = 0) Alors
                                        Ajouter_Axe (Ordre => 4)
                                FinSi
                        FinSi

                4 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Taille(AX) < LongMax) et (Taille(AX) < Distance (Origine(AX),
                                Origine(Axe_Support(AX))))

                        et (Date Modulo Frequence = 0) Alors
                                Ajouter_Arete
                        FinSi

                Autres => Rien

                FinSelon
        FinPour
FinPour
```

9 Feuille de Manioc ([Linnell & Arnoult 1985], Page 10)

9.1 Description informelle

Les axes d'ordre 1 croissent jusqu'à une certaine limite d'âge (ou de taille) (chacun développant à un certain moment de sa croissance de nouveaux aces d'ordre 1 à partir de la racine). Ils développent des axes secondaires, dont la croissance est limitée à un étage calculé, fonction de la distance de son point d'insertion à la racine de l'arbre). Ces axes sont insérés en alternance à gauche ou à droite de l'axe principal.

9.2 Pseudo Algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>

                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour


                        Si Date modulo Frequence = 0 Alors
                                Ajouter_Axe (Ordre => 1 ; Position => Racine)
                        FinSi


                        Si (Date > Age1) & (Date modulo Frequence2 = 0) Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => -30°)
                        SinonSi (Date > Age1) & (Date modulo Frequence3 = 0) Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => +30°)
                        FinSi

                2 =>
                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
```

```
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi

                                Si longueur (AR) < longueur (Morceau_D_Axe [Racine(AR), Noeud(Nd)]) &
                                        longueur (AR) < longueur_max Alors
                                                Ajouter_Arete
                                FinSi
                        FinPour

                FinSelon
        FinPour
FinPour
```

10 Feuille d'Igname ([Linnell & Arnoult 1985], Page 11)

10.1 Description informelle

L'axe principal croît jusqu'à une certaine limite d'âge ou de taille. A certains moments de son existence, il développe des axes secondaires à partir de la racine. Ceux-ci ont un angle d'insertion de plus en plus grand par rapport à l'axe d'ordre 1.

10.2 Pseudo Algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>

                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour
```

```
                        Si Date modulo Frequence = 0 Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => f (date) ; Position => Nd)
                        Fin Si

                2 =>

                        Si (Age (Ax) < Age_Max) Alors
                                Ajouter_Arete (Angle => Constant)
                        Fin Si

                        Pour toutes les aretes AR de AX Faire
                                Si Age < Age_Max Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                FinSi
                        FinPour

                FinSelon

        FinPour
FinPour
```

11 Feuille d'Arbre à Pain ([Linnell & Arnoult 1985], Page 12)

11.1 Description informelle

- Axe principal : croissance limitée à un âge donné. Les arêtes croissent de manière continue jusqu'à une longueur max. Insertion d'axes secondaires alternativement à gauche et à droite de l'axe, selon un angle donné.
- Axes d'ordre 2 : croissance limitée à un étage fixe. Les arêtes croissent de manière continue jusqu'à atteindre une longueur max. L'angle formé par les arêtes initiales avec l'axe principal évolue avec le temps. Insertion d'axes d'ordre 3, alternativement à gauche et à droite de l'axe, selon un angle donné.
- Axes d'ordre 3 : idem axe 2, sans insertion d'axes d'ordre 3.

11.2 Pseudo Algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>

                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si Date modulo Frequence1 = 0 Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => +A)
                        SinonSi Date modulo Frequence2 = 0 Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => -A)
                        FinSi

                2 =>
                        Si (longueur (AX) < distance(Racine (Parent(AX), Insertion (AX, Parent(AX)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                                -- Cette interpolation donne l'angle des arêtes de l'axe d'ordre 2 en fonction
                                -- de la distance a la racine de cet axe, le tout a l'etape N.
                                interpolation [distancemin, distancemax] -> [angle0, anglemax]

                                -- Cette interpolation donne l'angle d'une arete d'un axe d'ordre 2 en
                                -- fonction de son age et de sa distance a la racine de l'axe.
                                interpolation [age0, ageMax] -> [angle0, angleMax]
                        FinPour

                        Si Date modulo Frequence3 = 0 Alors
                                Ajouter_Axe (Ordre => 3 ; Angle => +B)
                        Sinon Si Date modulo Frequence4 = 0 Alors
                                Ajouter_Axe (Ordre => 3 ; Angle => -B)
                        FinSi

                3 =>
                        Si (longueur (AX) < distance(Racine (Parent(AX), Insertion (AX, Parent(AX)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour
                FinSelon
        FinPour
FinPour
```

12 Canne à Sucre ([Linnell & Arnoult 1985], Page 13)

12.1 Description informelle

- Axe principal : croissance limitée à un âge donné. Croissance des arêtes jusqu'à une taille limite. Insertion d'axes secondaires avec un angle donné. A partir d'une certaine date, insertion d'axes d'ordre 3 à une fréquence donnée.
- Axe d'ordre 2 : croissance limitée à une taille donnée. Croissance des arêtes jusqu'à une taille limite. Insertion d'axes d'ordre 4 alternativement à gauche et à droite de l'axe. L'angle des arêtes des axes d'ordre 2 est modifiée au cours du temps.
- Axe d'ordre 3 : Croissance limitée à un étage fixe. Insertion d'axes d'ordre 5 selon une fréquence donnée.
- Axe d'ordre 4 : Croissance limitée à un étage fixe. Croissance des arêtes jusqu'à une taille limite.
- Axe d'ordre 5 : Croissance limitée à un étage fixe. Croissance des arêtes jusqu'à une taille limite.

12.2 Pseudo Algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>

                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
```

```
                    Sinon
                            LongMax
                    FinSi
            FinPour

            Si Date modulo Frequence = 0 Alors
                    Ajouter_Axe (Ordre => 2 ; Angle => +A)
            FinSi

            Si (Date modulo Frequence2 = 0) & (Date > d) Alors
                    Ajouter_Axe (Ordre => 3 ; Angle => +A)
            Sinon Si (Date modulo Frequence3 = 0) & (Date > d) Alors
                    Ajouter_Axe (Ordre => 3 ; Angle => -A)
            FinSi

    2 =>
            Si (Age(Ax) < Age(Max)) Alors
                    Ajouter_Arete
            FinSi

            Pour toutes les aretes « AR » de AX Faire
                    Si Age(AR) < AgeMax Alors
                            Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                    Sinon
                            LongMax
                    FinSi
            FinPour

            Si Date modulo Frequence4 = 0 Alors
                    Ajouter_Axe (Ordre => 4 ; Angle => +B)
            Sinon Si Date modulo Frequence5 = 0 Alors
                    Ajouter_Axe (Ordre => 4 ; Angle => -B)
            FinSi

            Modif_Angle (Arete_Num ( (15 - Age(AX)) , 45° )
            Modif_Angle (Arete_Num ( (15 - Age(AX) + 1 ) , 0° )

    3 =>
            Si (Age(Ax) < Age(Max)) Alors
                    Ajouter_Arete
            FinSi

            Pour toutes les aretes « AR » de AX Faire
                    Si Age(AR) < AgeMax Alors
                            Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                    Sinon
                            LongMax
                    FinSi
            FinPour

            Si Date modulo Frequence = 0 Alors
                    Ajouter_Axe (Ordre => 5 ; Angle => +C)
            Sinon Si Date modulo Frequence' = 0 Alors
                    Ajouter_Axe (Ordre => 5 ; Angle => -C)
            FinSi

    4 =>
            Si (Age(Ax) < Age(Max)) Alors
                    Ajouter_Arete
            FinSi

            Pour toutes les aretes « AR » de AX Faire
                    Si Age(AR) < AgeMax Alors
                            Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                    Sinon
                            LongMax
                    FinSi
            FinPour

    5 =>
            Si (Age(Ax) < Age(Max)) Alors
                    Ajouter_Arete
            FinSi

            Pour toutes les aretes « AR » de AX Faire
                    Si Age(AR) < AgeMax Alors
                            Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                    Sinon
```

```
                                        LongMax
                                FinSi
                        FinPour

                FinSelon
        FinPour
FinPour
```

13 Feuille de Noisetier ([Linnell & Arnoult 1985], Page 20)

13.1 Description informelle

- Axe principal : Croissance de l'axe limitée à un âge donné. Croissance des arêtes limitée à un âge donné. Insertion d'axes secondaires alternativement à gauche et à droite de l'axe avec un angle donné.
- Axe d'ordre 2 : Croissance de l'axe limitée à un âge donné. Croissance des arêtes limitée à un âge donné. Insertion d'axes secondaires à droite de l'axe uniquement avec un angle donné selon une fréquence donnée. Modification de l'angle des arêtes de l'axe au cours du temps.

13.2 Pseudo Algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>

                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Date modulo Frequence1 = 0 Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => +A)
```

```
                        Sinon Si (Date modulo Frequence2 = 0) Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => -A)
                        FinSi

                2 =>
                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        -- On utilise une interpolation pour déterminer les angles des aretes des axes d'ordre
                        -- 2 à chaque etape de la croissance.
                        Pour toutes les aretes AR de AX Faire
                                interpolation [age0, AgeMax] -> [angle0, angleMax]
                        FinPour

                        Si Date modulo Frequence = 0 Alors
                                ajouter_axe (ordre => 2 ; angle => B)
                        FinSi

                FinSelon
        FinPour
FinPour
```

14 Feuille de noyer d'Amérique ([Linnell & Arnoult 1985], page 21)

14.1 Description informelle

- Axe principal : Croissance limitée à un âge donné. Croissance des arêtes limitée à un âge donné. Insertion d'axes secondaire à une fréquence donnée, alternativement à gauche et à droite de l'axe.
- Axe d'ordre 2 : Croissance limitée à un étage fixe. Modification de l'orientation des arêtes au cours du temps.

14.2 Pseudo Algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>

                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si (Date modulo Frequence1 = 0 Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => +A)
                        Sinon Si (Date modulo Frequence2 = 0) Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => -A)
                        FinSi

                2 =>
                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Modifier_Angle (Arete(N°2, AX), + f(longueur (AX))
                        Modifier_Angle (Arete(derniere, AX), - f(longueur (AX))

                FinSelon
        FinPour
FinPour
```

15 Feuille de Chêne ([Linnell & Arnoult 1985], Page 122)

15.1 Description informelle

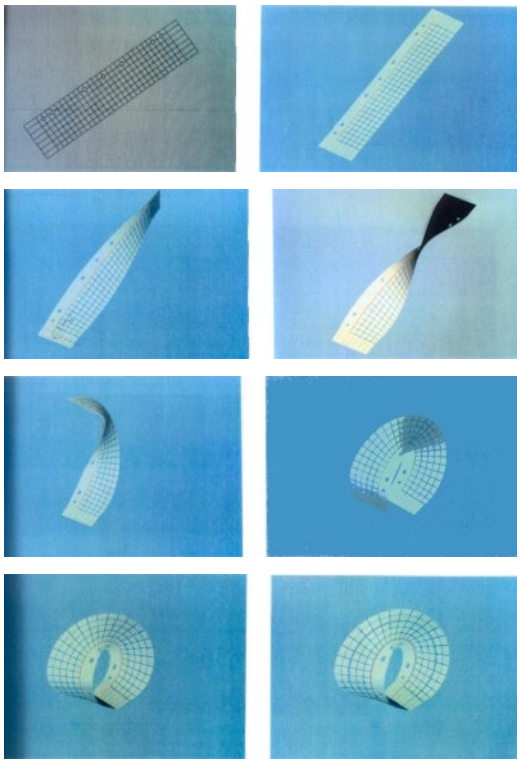
- Axe principal : Croissance limitée à un âge donné. Croissance des arêtes limitée à un âge donné. Insertion d'axes d'ordre 2 et 3 à une fréquence donnée, alternativement à gauche et à droite de l'axe.

- Axe d'ordre 2 : Croissance limitée à un étage calculé, fonction de la distance de l'insertion de l'axe sur l'axe principal avec la racine de la feuille.
- Axe d'ordre 3 : Croissance limitée à un étage calculé, fonction de la distance de l'insertion de l'axe sur l'axe principal avec la racine de la feuille (fonction qui limite plus la longueur que celle donnée pour l'axe d'ordre 2).

15.2 Pseudo Algorithme

```
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>

                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si Date modulo Frequence1 = 0 Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => +A)
                        SinonSi Date modulo Frequence2 = 0 Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => -A)
                        SinonSi Date modulo Frequence3 = 0 Alors
                                Ajouter_Axe (Ordre => 3 ; Angle => +B)
                        SinonSi Date modulo Frequence3 = 0 Alors
                                Ajouter_Axe (Ordre => 3 ; Angle => -B)
                        FinSi


                2 =>
                        Si (longueur (AX) < distance(Racine (Parent(AX), Insertion (AX, Parent(AX)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                3 =>
                        Si (longueur (AX) < distance(Racine (Parent(AX), Insertion (AX, Parent(AX)/2) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour


                FinSelon
        FinPour
FinPour
```

16 Terraz ([Terraz 1994], Page III-32 à III-33 – Planche C)



16.1 Description informelle

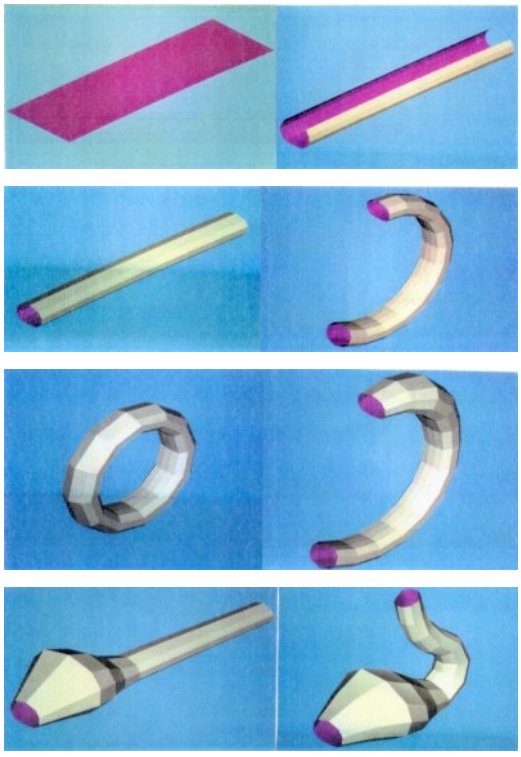
- Axe principal : Croissance limitée à un âge donné, croissance des arêtes limitée à un âge donné. Insertion d'axes secondaires perpendiculaires à l'axe d'ordre 1, à intervalles réguliers, de chaque côté de l'axe.
- Axe d'ordre 2 :
- Transformation

16.2 Pseudo Algorithme

```
-- Croissance
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>

                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Si date modulo frequence = 0 Alors
                                Ajouter_Axe (Ordre => 2 ; Angle => 90°)
                                Ajouter_Axe (Ordre => 2 ; Angle => -90°)
                        FinSi
```

```
                2 =>
                        Si longueur(AX) < longMax Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes AR de AX Faire
                                Si Age (AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [long0, longMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                FinSelon
        FinPour
FinPour

-- Transformation
-- Entree : les angles de la premiere et de la derniere arete des axes secondaires a l'etape N.
-- Sortie : l'angle de toutes les aretes secondaires a cette etape.

        Interpolation [Dmin, Dmax] -> [Angle(A0), Angle(An)]

-- NB : Dmin et Dmax sont les distances min et max separant une arete de la racine de l'axe principal.

-- Une deuxième interpolation permet de calculer l'angle d'une arete initiale d'un axe secondaire a une etape
quelconque a partir de son age.

        Interpolation [Age0, AgeMax] -> [AngleMin, AngleMax]
```

17 Terraz ([Terraz 1994], Pages III-32 à III-33 – Planche A)



17.1 Description informelle

- Axe principal
- Axe d'ordre 2

- Transformation

17.2 Pseudo Algorithme

```
-- Croissance
Pour chaque top d'horloge Faire
        Pour tous les axes « AX » de l'arbre Faire
                Selon Ordre(AX) Faire

                1 =>

                        Si (Age(Ax) < Age(Max)) Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes « AR » de AX Faire
                                Si Age(AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [Long0, LongMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                        Ajouter_Axe (Ordre => 2 ; Angle => 90°)
                        Ajouter_Axe (Ordre => 2 ; Angle => -90°)

                2 =>
                        Si longueur(AX) < longMax Alors
                                Ajouter_Arete
                        FinSi

                        Pour toutes les aretes AR de AX Faire
                                Si Age (AR) < AgeMax Alors
                                        Interpolation [Age0, AgeMax] -> [long0, longMax]
                                Sinon
                                        LongMax
                                FinSi
                        FinPour

                FinSelon
        FinPour
FinPour

-- Transformation
-- Photos 1 a 3
        -- Cette interpolation donne l'angle des aretes des axes secondaires selon leur age (toutes les aretes des
        -- axes secondaires ayant les memes angles).
        interpolation [Age1, Age3] -> [AngleMin, AngleMax]

-- Photos 3 a 5
        -- Cette interpolation donne l'angle des aretes de l'axe principal selon leur age (toutes les aretes de
        -- l'axe ayant le meme angle relatif).
        interpolation [Age4, Age5] -> [AngleMin, AngleMax]
```

Index

N

O

P

Q

S

T

V

Figures

Bibliographie

[Aliaswavefont 2002] Aliaswavefont. *Maya*, Aliaswavefont, Ed. 4.5, 2002.

[Bass et al. 1991] Bass l., Pellegrino R., Reed S., Sheppard S., & Szezur M. The Arch Model : Seeheim revisited. *User Interface Developper's Workshop,* 1991.

[Bauer 1979] Bauer M.A. Programming by Examples. *Articial Intelligence*, vol. 12, 1979, pp. 1-21.

[Bionatics 2001] Bionatics. *advanced plant modeling solutions using AMAP*, Bionatics, 2001.

[Bohm & Jacopini 1966] Bohm C. & Jacopini G. Flow diagram, Turing Machines and languages xith only two formation rules. *Communication of the ACM*, vol. 9, n° 5, 1966, pp. 141-153.

[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.

[Corcoran et al. 2002] Corcoran F., Demaine J., Picard M., Dicaire L.-G., & Taylor J. Inuit3D : An Interactive Virtual 3D Web Exhibition. *Conference on Museums and the Web 2002, Boston, MA*, April 17-20 2002.

[Coutaz 1987] Coutaz J. PAC, an Implementation Model for the User Interface. *IFIP TC13 Human-Computer Interaction (INTERACT'87),* Pub. North-Holland, *Stuttgart*, September 1987, pp. 431-436.

[Coutaz 1990] Coutaz J. *Interfaces Homme-Ordinateur, Conception et Réalisation*. Paris : Dunod Informatique, 1990, 455p.

[Coutaz 1992] Coutaz J. Interface Homme-Machine : un regard critique. *Journées d'Études AFCET : Interfaces Homme-Machine,* Pub. AFCET, *Paris*, 21 Octobre 1992, pp. 1-24.

[Coutaz & Nigay 1991] Coutaz J. & Nigay L. Seeheim et Architecture Multi-agent. *Interfaces Homme-Machine, Dourdan*, Décembre 1991.

[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.

[Cypher 1993] *Watch What I Do: Programming by Demonstration*. Ed. Cypher A., Cambridge, Massachusetts : The MIT Press, 1993, 604p.

[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.

[Depaulis 2000] Depaulis F. Représentation et correction de programme dans les systèmes de programmation sur exemple. *RJC'IHM 2000, Ile de Berder*, 3-5 Mai 2000 2000, pp. 27-30.

[Depaulis, Maiano, & Texier 2002] Depaulis F., Maiano S., & Texier G. DTS-Edit : an Interactive Development Environment for Structured Dialog Applications. *CADUI'02,* Pub. Kluwer Academics, *Valenciennes (France)*, May, 15-17th 2002, pp. 75-82.

[Discreet 2002] Discreet. *3D Studio Max*, Discreet, Ed. 5.0, 2002.

[Duke & Harrison 1993] Duke D.J. & Harrison M.D. Abstract Interaction Objects. *Computer Graphics Forum*, vol. 12, n° 3, 1993, pp. 25-36.

[Eyrolles et al. 1989] Eyrolles G., Viennot X., Janney N., & Arquès D. Combinatorial Analysis of Ramified Patterns and Computer Imagery of Trees. *Computer Graphics*, vol. 23, 1989, pp. 31-40.

[Fekete 1996] Fekete J.-D. *Un modèle multicouche pour la construction d'applications graphiques interactives*. Doctorat d'Université (PhD Thesis) : LRI, Université Paris-Sud, Orsay, 1996, 203p.

[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. *MI-CAD,* Ed. Hermès, *Paris, 21-25 Mars*, 1988, pp. 281-296.

[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.

[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.

[Girard 2000] Girard P. *Ingénierie des systèmes interactifs : vers des méthodes formelles intégrant l'utilisateur*. Habilitation à diriger les recherches : LISI/ENSMA, Université de Poitiers, Poitiers, 2000, 92p.

[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.

[Girard & Pierra 1994] Girard P. & Pierra G. *End-User Programming Environments : an Up-Dated Taxonomy*. LISI-ENSMA, 1994 94002.

[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.

[Goldberg 1984] Goldberg A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[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.

[Guittet & Pierra 1993a] 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 1993a, pp. 151-156.

[Guittet & Pierra 1993b] Guittet L. & Pierra G. *La notion d'interacteur : un concept unificateur pour la conception modulaire d'une application graphique interactive de conception technique*. Laboratoire d'Informatique Scientifique et Industrielle de l'Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, 1993b 93008.

[Guittet & Pierra 1993c] Guittet L. & Pierra G. *Organisation modulaire et specification d'une application de CAO : hiérarchie d'Interacteurs dans le système NODAO*. Laboratoire d'Informatique Scientifique et Industrielle de l'Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, 1993c 93011.

[Gwenola & Donokian 2000] Gwenola T. & Donokian S. Virtual humans animation in informed urban environments. *Computer Animation 2000 (CA'00), Philadelphia, Pennsylvania*, May 03-05 2000, pp. 112.

[Hachette 2002] Hachette. *Encyclopédie Multimédia Hachette*. Hachette, 2002.

[Halbert 1984] Halbert D. *Programming by Example*. PhD Thesis : University of California, Berkeley, 1984, 121p.

[Halbert 1993] Halbert D. *SmallStar : Programming by Demonstration in the Desktop Metaphor*. *in* Watch What I Do : Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusuetts : The MIT Press, 1993, pp. 102-123.

[Harrison & Duce 1994] Harrison M.D. & Duce D.A. *A review of formalisms for describing interactive behaviour*. University of York, January 7 1994.

[Hegron & Arnaldi 1992] Hegron G. & Arnaldi B. Computer Animation: Motion and Deformation Control. *Eurographics'92, Cambridge, USA*, September 7-11 1992.

[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),* Eds. D.J. Duke & A. Puerta, Ser. SpringerComputerScience, Eds. Ser. W. Hansmann, W.T. Hewitt & W. Purgathofer, Pub. Springer-Verlag, *Universidade do Minho, Braga, Portugal*, 2-4 June 1999, pp. 205-224.

[Kurlander & Feiner 1988] Kurlander D. & Feiner S. Editable Graphical Histories. *IEEE Symposium on Visual Languages,* Pub. IEEE, *Pittsburg*, October 1988 1988, pp. 127-133.

[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.

[Kurlander & Feiner 1993] Kurlander D. & Feiner S. *A History of Editable Graphical Histories*. *in* Watch What I Do : Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993, pp. 405-413.

[Lieberman 1993a] Lieberman H. *Mondrian: a Teachable Editor*. *in* Watch What I Do: Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993a, pp. 341-360.

[Lieberman 1993b] 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, 1993b, pp. 49-66.

[Lieberman 2001] Lieberman H. *Your Wish is my command*. Morgan Kaufmann, ISBN 1-55860-688-2, 2001, 416p.

[Lienhardt 1987] Lienhardt P. *Modélisation et évolution de surfaces libres*. Informatique, Université Louis Pasteur, Strasbourg, 1987, 145p.

[Linnell & Arnoult 1985] Linnell T. & Arnoult J. *Plantes utiles du monde entier*. Fernand Nathan, 1985, 195p.

[Lintermann & Deussen 1999] Lintermann B. & Deussen O. Interactive Modeling Of Plants. *IEEE Computer Graphics & Applications*, vol. 19(1), 1999, pp. 56-65.

[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.

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

[Mandelbrot 1975] Mandelbrot B. *Les Objets Fractals - Forme, Hasard et Dimension*. Flammarion, 1975, 190p.

[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.

[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.

[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.

[Mills 1975] Mills H. The new Math of Computer Programming. *Communivation of the ACM*, vol. 18, n° 1, 1975, pp. 74-82.

[Modugno & Myers 1993] Modugno F. & Myers B.A. *Graphical Representation and Feedback in a PbD System*. *in* Watch What I Do : Programming by Demonstration, Ed. A. Cypher, Cambridge, Massachusetts : The MIT Press, 1993, pp. 415-422.

[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.

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

[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.

[Nigay 1994] Nigay L. *Conception et Modélisation Logicielle des Systèmes Interactifs : Application aux Interfaces Multimodales*. Doctorat d'Université (PhD Thesis) : CLIPS/IMAG, Université Joseph Fourier, Grenoble, 1994.

[Nigay & Coutaz 1991] Nigay L. & Coutaz J. Building User Interfaces: Organizing Software Agents. *ESPRIT'91 Conference,* 1991.

[Nigay & Coutaz 1993] Nigay L. & Coutaz J. A Design for MultiModal Systems : Concurrent Processing and Data Fusion. *INTERCHI'93,* Eds. S. Ashlung, K. Mullet, A. Henderson, E. Hollnagel & T. White, Pub. ACM New York, *Amsterdam*, 1993, pp. 172-178.

[Norman 1986] Norman D. *User Centered System Design*. Lawrence Erlbaum Associates, ISBN 0898598729, 1986.

[O'Donnell & Olson 1981] O'Donnell T.J. & Olson A.J. GRAMPS : A graphics language interpreter for real-time, interactive, three dimensional picture editing and animation. *Computer Graphics*, 1981, pp. 133-142.

[Olsen 1998] Olsen D.R. *Developping User Interfaces*. San Francisco, Californie : Morgan Kaufmann Publishers, ISBN 1-55860-418-9, 1998, 414p.

[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.

[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.

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

[Patry 1999] 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, 1999, 199p.

[Pfaff 1985] *User Interface Management Systems, Proceedings of the Workshop on User Interface Management Systems held in Seeheim*. Ed. Pfaff G.E., Berlin : Springer-Verlag, 1985.

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

[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*, June 7-9 1995, pp. 362-370.

[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.

[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.

[Prusinkiewicz & Hammel 1996] Prusinkiewicz P. & Hammel M. *Visual model of plant development*. *in* Handbook of Formal Languages, , Springer Verlag, 1996,

[Reeves 1983] Reeves W.T. Particle Systems - A technique for Modeling a Class of Fuzzy Objects. *Computer Graphics*, vol. 17, 1983, pp. 359-376.

[Reffye et al. 1988] Reffye P.D., Edelin C., Françon J., Jaeger M., & Puech C. *Plant models faithful to botanical structure and development*. *in* Proceedings of the 15th annual conference on Computer graphics and interactive techniques, , 1988, pp. 151--158.

[Reynolds 1987] Reynolds C.W. Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics*, vol. 21, 1987, pp. 25-34.

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

[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.

[Softimage 2002] Softimage. *Softimage XSI*, Softimage, Ed. 2.0, 2002.

[Sutherland 1963] Sutherland I.E. Sketchpad : A man machine graphical communication system. *Proceedings-Spring Joint Computer Conference,* Pub. IFIP/AFIPS, 1963, pp. 329-346.

[Szilard & Quinton 1979] Szilard A.L. & Quinton R.E. An interpretation for D0L-systems by computer graphics. *The Science Terrafin,* April, 8-13 1979.

[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.

[Terraz 1994] Terraz O. *Programmation de métamorphoses d'objets surfaciques et volumiques*. Université Louis Pasteur, Strasbourg, 1994, 250p.

[Texier 2000] Texier G. *Contribution à l'ingéniérie des systèmes interactifs : Un environnement de conception graphique d'applications spécialisées de conception*. Université de Poitiers, Poitiers, 2000, 230p.

[Texier, Depaulis, & Guittet 2001] Texier G., Depaulis F., & Guittet L. End-User Class Definition in CAD Systems. *2001 IEEE Symposia on Human-Centric Computing Languages and Environments,* Ed. IEEE, Pub. Entergraphica, *Stresa, Italy*, September 5-7 2001 2001, pp. 180-187.

[Texier & Guittet 1999a] Texier G. & Guittet L. Dialogue+Gadget=Diaget. *Onzièmes journées sur l'ingénierie de l'Interaction Homme-Machine,* Pub. Cépadues-Editions, *Montpellier France*, 22-26 Novembre 1999a, pp. 70-77.

[Texier & Guittet 1999b] 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 Academics, *Louvain-la-Neuve, Belgique*, 21-23 October 1999b, pp. 231-244.

[Texier, Guittet, & Girard 2001] Texier G., Guittet L., & Girard P. The Dialog Toolset: a new way to create the dialog component. *Universal Access in HCI,* Ed. C. Stephanidis, Pub. Lawrence Erlbaum Associates, *New-Orleans, Louisiana, USA*, August 5-10, 2001 2001, pp. 200-204.

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

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

[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.

[Woods 1970] Woods W. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM*, vol. 13, n° 10, 1970, pp. 591-606.

[Zeltzer 1985] Zeltzer D. Toward an integrated view of 3-D computer animation. *CG&A*, 1985, pp. 87-101.