N° d'ordre : 2034

THESE

Présentée à

L'UNIVERSITE DES SCIENCES ET DES TECHNOLOGIES DE LILLE

Pour obtenir le titre de

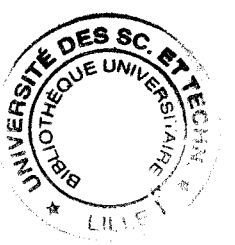
Docteur

en Productique, Automatique et Informatique Industrielle

par

Frédéric Van de Veire

Maître ès Science EEA



La Simulation et l'Animation Modulaire d'Algorithmes en Langage Objet

Soutenue le 19 septembre1997 devant la commission d'examen composé de Messieurs :

P. Vidal	Président	Professeur à l'U.S.T.L.
C. Kolski	Rapporteur	Professeur à l'U.V.H.C.
S. Wegrzyn	Rapporteur	Professeur à l'Ecole Polytechnique de Silésie de Pologne
J.M. Toulotte	Directeur de recherche	Professeur à l'U.S.T.L.
P. A. Szmal	Membre	Professeur à l'Ecole Polytechnique de Silésie de Pologne
R. Ikni	Membre	Maître de Conférence à l'Université du Littoral

SOMMAIRE

Liste des Figures

Liste des Tableaux

REMERCIEMENTS

Le travail présenté dans ce mémoire a été effectué au Centre d'Automatique de l'Université des Sciences et Technologies de Lille, dirigé par Monsieur le Professeur Pierre VIDAL dont je tiens à remercier pour l'accueil qu'il m'a réservé au sein de son laboratoire et de l'honneur qu'il me fait en acceptant d'être le président de mon jury de thèse.

Toute ma reconnaissance et mes sentiments respectueux vont à Monsieur le Professeur Jean-Marc TOULOTTE pour la confiance et le soutien qu'il m'a témoigné tout au long de mes travaux.

Je remercie également Messieurs : Christophe KOLSKI, Professeur à l'Université de Valenciennes et du Hainaut-Cambrésis et Stéfan WEGRZYN, Professeur à l'Ecole Polytechnique de Silésie, Directeur de l'Institut de l'Informatique Théorique et Appliqué de Pologne pour l'honneur qu'ils me font en acceptant de juger ce travail et d'en être rapporteurs.

Mes remerciements vont également à Messieurs : Przemyslaw A. SZMAL, Professeur à l'Ecole Polytechnique de Silésie de Pologne et Rachid IKNI, Maître de Conférence à L'Université du Littoral, pour l'intérêt qu'ils me témoignent en acceptant de participer à mon jury de thèse.

Enfin, je suis très reconnaissant pour l'aide que m'ont apportée tous les membres du laboratoire d'Automatique et de la sympathie qu'ils m'ont témoignée.

Je dédie cette thèse principalement à mes parents et à ma femme que j'aimerais remercier pour leur patience, leur support moral, leur compréhension et leurs encouragements sans cesse renouvelés, qui furent de première importance tout au long de ces longues années.

INTRODUCTION GÉNÉRALE

L'animation d'algorithmes est le processus d'abstraction de données, d'opérations, et de sémantique de programmes informatiques sous la forme de représentations et d'animations graphiques. Un utilisateur peut alors reconnaître visuellement les comportements et les relations complexes entre ces informations. Il est en fait généralement accepté que les techniques de visualisation graphique ont un rôle à jouer durant la première phase de conception d'un logiciel. Mais ce qui devient, également, plus évident est que la visualisation a pénétré tous les stades de développement informatique incluant la phase d'implantation, d'exécution et de test. Il est souvent clamé que ce type d'environnement peut améliorer la compréhension des programmes, servir de moyen de communication, de déboguage et d'évaluation et souvent augmenter la productivité, la maintenance et la réutilisabilité. A cet effet, il y a eu une prolifération de conceptions de bibliothèques graphiques, de débogueurs, d'éditeurs basés sur des structures graphiques grammaticales, des systèmes de programmation graphique, des langages textuels visuels, des outils pour la simulation interactive et l'animation de données ou de code du programme. Par conséquent, la frontière traditionnelle entre l'analyse, la conception, l'implémentation, le test et l'installation réelle devient floue et différents types d'utilisateurs interagissent et contribuent à un projet avec chacun leur propre perception.

Enormément de systèmes d'animation ont été développés ces dernières années, toutefois relativement peu de travaux ont abouti à l'obtention d'un environnement à la fois

convivial, interactif et homogène permettant de présenter graphiquement et d'animer la plupart des informations issues d'un programme élémentaire ou large.

Nous avons ainsi développé des outils et les concepts nécessaires à la détermination d'un modèle de visualisation d'algorithme et des différents composants qui le constituent. Nous aboutissons de cette façon au développement d'un environnement combinant des outils de visualisation de programme et de programmation visuelle permettant respectivement de visualiser les informations des algorithmes élémentaires, et de créer et visualiser, à partir de ceux-ci ou de nouveaux modules constituant des composants logiciels indépendants, de larges programmes.

Ainsi, la convivialité de l'environnement obtenu se caractérise principalement par la présentation de tout algorithme élémentaire ou module, de son code, de ses données, de ses expressions et de son comportement dans des fenêtres graphiques autonomes. Cette convivialité permet alors de gérer l'espace de travail en donnant à l'utilisateur toute liberté de visualiser n'importe quelle information susceptible de l'intéresser. En outre, la remise à jour de ces fenêtres s'effectue de manière simple et automatique, au travers de l'utilisation d'un concept de dépendance, permettant de garder une cohérence entre le comportement des objets graphiques animés et le programme. Cette notion va nous permettre d'obtenir et de conserver une simulation interactive, c'est-à-dire d'autoriser l'utilisateur à modifier toutes les informations issues d'un programme ou d'un module ainsi que tous les paramètres de visualisation du système. Nous avons, également, introduit et adapté un modèle d'architecture graphique appelé P.Os.T. qui nous permet de concevoir les représentations et les animations des informations d'un algorithme ainsi que des modules. Cette construction s'effectue par l'intermédiaire d'une bibliothèque prédéfinie d'images et d'animations ou en en définissant de nouvelles par l'entremise des outils conviviaux et interactifs développés autour de celui-ci.

Ainsi après avoir établi un programme de manière textuelle ou visuelle, l'utilisateur peut visualiser l'exécution du code d'un algorithme élémentaire ou d'un module employé dans un large programme, visualiser l'état de ces données, etc., de manière graphique et modifier au cours de l'exécution du programme ces différentes informations.

Aussi, l'association de la programmation visuelle et de la visualisation de programme aboutit à un environnement homogène qui peut être utilisé par les enseignants et les chercheurs, pour comprendre le fonctionnement interne des programmes, pour en développer de nouveaux ou explorer de nouvelles variantes et évaluer les performances de ceux qui existent.

L'environnement et les outils conçus sont présentés dans ce mémoire structuré en six chapitres.

Le premier chapitre est la présentation du domaine de notre étude et nous le définissons par l'intermédiaire de son vocabulaire et sa problématique. Nous présentons également succinctement nos contributions en mettant en avant les critères de convivialités adoptés.

Au second chapitre, un modèle général pour construire un logiciel de visualisation graphique interactif et plus particulièrement de visualisation d'algorithmes a été défini. Ce modèle fournit une base conceptuelle, qui nous permet de déterminer les principaux éléments composant un tel logiciel et par la même de répertorier et d'évaluer pour chacun de ces constituants les différentes caractéristiques et techniques mises en œuvre dans de précédents systèmes de même type.

De cette étude nous avons pu déterminer les avantages et inconvénients des différents logiciels de visualisation d'algorithmes, nous permettant de déterminer les caractéristiques de notre propre système. Celui-ci est présenté au troisième chapitre où nous décrivons plus particulièrement les trois composantes basiques qui le constituent, en l'occurrence :

- l'interface et les différents aspects de visualisation d'un algorithme,

- la méthode de gestion des événements de la simulation transposant une exécution en une animation et qui est basée sur la notion de dépendance associée au modèle M.V.C. (Modèle, Vue, Contrôleur) de Smalltalk et enfin,

- le modèle d'animation graphique basé sur le modèle P.Os.T. (Présentation, Objet de simulation, Traducteur) [Mos 94a].

Nous présentons au quatrième chapitre l'outil de programmation visuelle permettant la construction graphique d'algorithmes. Ceux-ci seront constitués de modules autonomes et élémentaires, présentés sous forme d'icône, et seront connectés interactivement par l'intermédiaire de la souris créant un réseau de relations simples et cohérentes entre les différentes données des modules employés. Nous décrivons, plus explicitement, au cours de ce chapitre, la représentation de ces modules, leur gestion graphique et structurelle et les différentes formes de visualisation d'informations et d'interaction qui leur sont associées.

Enfin, avant de présenter nos conclusions et nos perspectives, nous faisons au cinquième chapitre une évaluation du système proposé par l'intermédiaire de quatre exemples d'animations et démontrons son intérêt.

Cependant, désirant un environnement ouvert et homogène, utilisable par le plus grand nombre d'utilisateurs possible, nous avons développé en collaboration avec Monsieur Przemylaw A. Szmal, Professeur à l'Institut Informatique de l'Université Technique de Silésie de Pologne, sous Smalltalk, un traducteur Pascal de haut niveau. Celui-ci nous permet d'écrire les algorithmes en employant indifféremment la syntaxe du langage Pascal ou Smalltalk par différenciation automatique des entêtes.

Nous parlerons ainsi tout au long de cette thèse d'algorithmes employant l'un ou l'autre de ces langages.

Il est à noter que, si nous avons choisi de présenter cet outil en annexe A, ce n'est pas qu'il constitue, à nos yeux, un outil mineur dans notre environnement, car il permet à des utilisateurs ne connaissant pas les concepts d'objets et plus particulièrement la syntaxe du langage Smalltalk d'utiliser notre système. Il ne fait toutefois pas directement partie des éléments conventionnels d'une animation modulaire d'algorithmes.

CHAPITRE I

DÉFINITIONS ET PROBLÉMATIQUE DE LA VISUALISATION D'ALGORITHMES ET DE LA PROGRAMMATION VISUELLE

La visualisation durant cette dernière décennie est devenue un outil indispensable [MCC 87]. Dans la médecine, la physique, la météorologie, l'ingénierie, et bien d'autres domaines scientifiques l'emploi d'images sert à montrer les comportements dynamiques de nombreuses données obtenues par observations ou générées par simulation.

L'intérêt de telles visualisations est suscité par le fait qu'une seule image est capable de transporter une grande somme d'informations liés au phénomène physique considéré. Elles peuvent également s'appuyer sur la similarité de leur représentation avec le système réel ce qui a une influence importante pour leur compréhension. Concrètement, l'utilisation de visualisations abstraites des informations d'un système et l'animation utilisent les capacités de l'esprit humain à reconnaître visuellement des comportements et des relations complexes entre plusieurs données.

Aussi, le pouvoir d'abstraction inhérent à la présentation visuelle et l'aptitude innée de l'être humain à interpréter de très nombreuses informations visuelles sont les clés de la réussite d'une visualisation scientifique [SHU 88].

Ces arguments ont amené différents chercheurs à employer les techniques de visualisations graphiques dans bien d'autres domaines et en particulier en informatique dans l'aide à la programmation, au déboguage et à la compréhension des programmes.

Ceci a été motivé par le fait que les algorithmes en informatique sont écrits en utilisant des langages de programmation évolués et, par là même, ils ont recours à des syntaxes et des structures de données spécifiques. Il est donc très difficile pour de simples utilisateurs de constituer de nouveaux programmes ou de les comprendre quand ils sont vus sous leur forme textuelle traditionnelle [LEW 87].

Nous pouvons également remarquer, qu'aujourd'hui, l'essor de la vente des ordinateurs personnels est lié aux très nombreux logiciels vendus avec ceux-ci et non plus à la volonté de l'utilisateur de créer sa propre application. La majorité de ces acheteurs ne savent et ne veulent plus apprendre de langage de programmation complexe. Ceci a un effet pervers car, ne sachant pas modifier les programmes qu'ils emploient, ils sont obligés de faire appel à des logiciels pas toujours adaptés à leur propre problème.

Pour rendre la tâche de programmation plus accessible à ces utilisateurs, une des approches est d'employer le graphisme comme langage de programmation. Cette forme de programmation est appelée "programmation visuelle" ou "programmation graphique". Ainsi, l'élaboration de nouveaux algorithmes sous forme graphique aide l'utilisateur à se détacher des contraintes syntaxiques d'un langage de programmation et pour un utilisateur expérimenté à être plus productif en ne réécrivant plus d'algorithmes élémentaires.

Une autre approche est l'introduction de visualisations graphiques, statiques et dynamiques des données et du programme pour aider l'utilisateur à comprendre et à reconnaître visuellement les comportements et les relations qui peuvent exister entre plusieurs informations issues des algorithmes exécutés. D'ailleurs, si nous prenons comme exemple deux livres majeurs qui ont été écrits dans le cadre de l'enseignement des algorithmes [STA 80, SED 83] ceux-ci font appel, pour décrire les algorithmes qu'ils étudient, à des vues qui représentent les structures de données utilisées sous forme de dessin à deux dimensions accompagnant le texte de base.

En fait, nous pensons que la représentation textuelle des données comme du programme peut obscurcir la compréhension des fonctionnalités d'un algorithme. En ajoutant des animations graphiques représentant le comportement des données et du programme lors de son exécution à la visualisation textuelle traditionnelle nous aurons une meilleure transmission du sens, de la méthodologie et du but atteint par l'algorithme.

Aussi, l'association de ces deux approches aboutit à un environnement de programmation homogène qui peut être utilisé par les enseignants et les chercheurs qui reçoivent ainsi de l'aide de trois manières différentes pour :

- **Développer de nouveaux programmes,** car la programmation visuelle et l'animation aident les chercheurs et les concepteurs de programmes à explorer plus facilement de nouvelles variantes d'algorithmes existants. D'autre part, la visualisation graphique aide à illustrer les comportements et les caractéristiques d'un programme durant son développement initial et a pour conséquence de promouvoir la découverte de solutions alternatives aux problèmes étudiés.

- **Comprendre la fonctionnalité des programmes,** car l'animation aide les programmeurs à expliquer le fonctionnement interne de programmes qui sont difficiles à comprendre. En particulier, l'enseignement bénéficie de représentation graphique du programme pour accompagner la visualisation textuelle traditionnelle.

- **Evaluer des programmes existants,** car l'animation aide les programmeurs à contrôler les performances des programmes. En permettant, par exemple, de comparer deux algorithmes qui traitent les mêmes données, de visualiser des conditions infranchissables ou une file de tâches et de détecter une mauvaise gestion d'un périphérique.

I.1 - DEFINITIONS

Pour mieux introduire notre logiciel et les approches que nous avons suivies au cours de cette thèse, nous allons définir le domaine de notre étude à travers son vocabulaire et les définitions qu'il engendre.

I.1.1 - Algorithme

Le "petit Robert" définit "***algorithme***" comme étant "un ensemble de règles opératoires propres à un calcul". Ou, d'une façon plus générale, un algorithme est la description logique et chronologique d'une suite d'opérations permettant d'obtenir un résultat bien spécifié sous hypothèse de conditions initiales déterminées ; c'est le modèle abstrait d'un problème concret. Il s'agit en fait d'une méthode systématique, susceptible d'une réalisation mécanique, pour résoudre un problème donné.

Il faut, ainsi, remarquer que la notion d'algorithme n'est pas spécifique à l'informatique. Certains algorithmes décrivent toutes sortes de tâches quotidiennes. Quelques exemples sont donnés ci dessous. Les forces agissantes qui exécutent une tâche sont appelées des processeurs et sont le fait d'êtres humains, ou d'ordinateurs (Cf. tableau 1.1). Un processeur obéit aux ordres que l'algorithme lui décrit et les traduit en actions. L'exécution d'un algorithme implique l'exécution de chacune de ses étapes constitutives.

Processus	***Algorithme***	***Etapes typiques d'un algorithme***
Tricoter un pull	Explication du tricot	Une maille à l'endroit, une maille à l'envers
Construire un modèle d'avion	Instruction d'assemblage	Coller le panneau A au support B
Faire un gâteau	Recette	Prendre 3 oeufs, battre jusqu'à homogénéité
Faire un vêtement	Patron	Ajuster et coudre le côté

***Tableau 1.1** : Détermination pour un processus donné de l'algorithme et des étapes typiques lui afférant.*

Ainsi, les algorithmes sont fondamentaux, ils sont à la fois indépendants du langage dans lequel ils sont énoncés et de l'ordinateur qui les exécute. Prenons pour nous aider une analogie avec la vie quotidienne. Une recette de tarte aux fruits peut être écrite en anglais ou en français. Pourtant, quelle que soit la langue, l'algorithme est fondamentalement le même.

Un langage de programmation n'est qu'un moyen pratique d'énoncer un algorithme, et un ordinateur est un simple processeur qui l'exécute. Le langage de programmation et l'ordinateur ne sont que les moyens, le but étant l'exécution de l'algorithme et le déroulement du traitement correspondant.

Par conséquent, les algorithmes en informatique sont la retranscription d'une méthodologie de résolution d'un problème spécifique dans un langage donné (du type Fortran, Pascal, C). Ainsi, comme nous l'avons dit précédemment, les programmes obtenus sont souvent difficiles à comprendre ; car les structures de données employées ne sont qu'une représentation informatique, dépendantes du langage de programmation utilisé et sont donc éloignées de l'image réelle du problème posé. Nous percevons également que la visualisation d'un algorithme ne peut se limiter à la simple visualisation des données et de l'état de contrôle du programme ; mais elle doit être une représentation concrète du problème, aboutissant à une représentation abstraite ou de synthèse du programme.

I.1.2 - La visualisation de programmes.

La visualisation de programmes, encore appelée animation d'algorithmes, concerne tous les systèmes qui permettent à l'utilisateur de visualiser les informations d'un programme sous la forme d'une, deux ou plusieurs dimensions de manière statique ou dynamique. Nous trouvons ainsi les systèmes de "visualisation de données" qui montrent l'image de l'état des données du programme. De même, les systèmes de "visualisation du code" illustrent l'image textuelle réelle du programme en lui ajoutant des marques graphiques ou en le convertissant sous une forme graphique (par exemple, sous forme d'organigrammes). Les systèmes qui illustrent "l'algorithme" utilisent le graphisme pour montrer concrètement comment le programme opère. Cette forme de visualisation, comme nous le remarquerons ultérieurement, est différente de la visualisation des données et du code car elle ne correspond pas directement à la représentation des variables du programme ; et les modifications de cette image peuvent ne pas correspondre à un morceau spécifique du code.

Enfin, la visualisation dynamique concerne tous les systèmes qui peuvent montrer une animation du programme lors de son exécution, tandis que les systèmes de visualisation statique sont limités aux images du programme à un instant donné.

Mais ces différentes visualisations ne peuvent être généralement établies que pour des programmes de tailles réduites. Ceci est dû au fait que des algorithmes complexes font appel à plusieurs processus qu'il est difficile de visualiser sous la forme d'une unique image.

I.1.3 - Les modules.

Une des possibilités pour résoudre le problème de visualisation lié à la complexité est de s'inspirer des paradigmes de la programmation classique en ce domaine.

La première notion introduite dans les langages de programmation fut celle de la décomposition d'un programme sous forme de sous-programmes (ou encore procédure, routine, fonction, ...) représentant généralement des tâches répétitives. Les sous-programmes furent, entre autres, l'un des premiers moyens pour, d'une part, structurer un programme et d'autre part, effectuer un masquage d'informations. En effet, un programme peut utiliser un sous-programme écrit par un autre programmeur en ne connaissant que le nom et les paramètres de ce sous-programme. Néanmoins, Timothy Budd [BUD 91] met en évidence au moins une des raisons essentielles de l'insuffisance du niveau d'abstraction des sous programmes, en identifiant des problèmes de visibilité de certains identificateurs qui ne sont

que partiellement résolus (problème des variables globales et locales). C'est pour résoudre ce problème que la notion de module a été introduite.

Un algorithme qui peut être intégré à un autre algorithme est un module (dans certains langages de programmation ce peut être une procédure, une routine, une sous routine ou une fonction). Un module est un algorithme dit élémentaire et peut être conçu indépendamment du contexte dans lequel il doit être utilisé.

En fait, pour répondre effectivement au problème de masquage de l'information, un module est dissocié en une partie visible "dite publique" et une partie "privée". La partie publique, appelée couramment interface, est accessible de l'extérieur du module tandis que la partie privée n'est accessible que de l'intérieur du module. Les techniques pour déterminer les modules et leurs interfaces sont clairement expliquées par David Parnas [PAR 72].

Les avantages que comporte l'utilisation de modules peuvent être résumés de la manière suivante :

- Un algorithme est un composant ou module de tout autre algorithme plus important et peut donc être conçu indépendamment l'un de l'autre, simplifiant la procédure de conception.

- Pour intégrer un module à un algorithme, il suffit de savoir ce que fait l'algorithme et non comment il le fait.

- De même que les modules simplifient la conception des algorithmes, ils simplifient aussi leur compréhension (car nous pouvons agir uniquement sur une partie d'un programme en occultant certaines opérations).

- Une fois qu'un module a été conçu, il peut être intégré à tout algorithme qui en a besoin. Il est, de ce fait, utile de construire une bibliothèque de modules, tels que des modules de tri, de résolution d'équation, de traitement de matrice, etc.

Ainsi la modularité répond à des objectifs de réutilisabilité, d'extensibilité et de composabilité. Son utilisation va nous permettre de visualiser par parties un programme par l'intermédiaire d'une interface appropriée. Mais, pour bénéficier de tous les avantages de la modularité, il faut que celle-ci puisse nous permettre de composer de nouveaux programmes de manière interactive et conviviale ; c'est ce que va nous permettre la programmation visuelle en plus de l'aide à la programmation.

I.1.4 - La programmation visuelle.

La programmation visuelle est un concept entièrement différent de la visualisation de programme.

Myers [MYE 90a] a donné une bonne définition pour distinguer ces deux domaines de visualisation : "Dans la programmation visuelle, les images graphiques sont utilisées pour créer le programme lui-même, mais dans la visualisation de programme, le programme est spécifié d'une manière textuelle et les graphiques sont utilisés pour illustrer plusieurs aspects du programme lors de son exécution". Malheureusement, dans le passé, plusieurs systèmes de visualisation de programme ont été incorrectement qualifiés de programmation visuelle (comme dans [GRA 85]).

Malgré tout, ces deux domaines sont complémentaires car ils emploient tous les deux des techniques de visualisation qui ont déjà un impact significatif sur l'environnement des langages de programmation [AMB 89].

On peut dire que la programmation visuelle fait référence à tout système qui permet de spécifier un programme d'une façon multidimensionnelle. Les langages de programmation textuels classiques ne sont pas considérés comme bidimensionnels car ils peuvent être analysés séquentiellement, symbole après symbole [MYE 90a]. De même, les langages de programmations tels que : Visual Basic, Visual C++, ou Delphi ne peuvent être considérés comme des langages de programmation visuelle malgré leur nom. En fait, ces systèmes ne sont que des langages textuels qui emploient un constructeur graphique du type GUI (Graphical User-Interface) pour programmer plus facilement des interfaces graphiques.

Cependant la programmation visuelle comprend les langages de programmation graphique utilisant les organigrammes pour créer les programmes mais elle ne comprend pas les systèmes qui utilisent les langages de programmations conventionnelles (dit linéaires) pour définir les images (telles que Sketchpad [SUT 63], CORE, PHIGS, Postscript [ABO 85], la Macintosh Toolbox [APP 85], ou la X-11 Window Manager Toolkit [MCC 88]).

En fait, un langage visuel est un langage qui manipule des informations visuelles, qui permet une interaction visuelle, ou qui permet de programmer à l'aide d'expressions visuelles. Ces langages peuvent être classés, en fonction du type et du degré d'expression visuelle utilisés en :

- langages à base d'icônes [GLI 92, NOR 90],
- langages à base de formes [YAN 94, AMB 90] et en

- langages à base de diagrammes (le plus souvent de graphes orientés) [PAP 95, SCH 91].

Ainsi, les environnements de programmation visuelle fournissent les éléments graphiques ou iconiques qui peuvent être manipulés interactivement par l'utilisateur en fonction d'une grammaire spatiale spécifique pour construire un programme [GOL 90].

Cependant, il existe dans la programmation visuelle plusieurs manières d'employer les aspects visuels pour définir un programme. Ceux-ci sont en fait liés aux différents paradigmes de programmation utilisés.

I.1.4.1 - Paradigmes de la programmation visuelle

Les paradigmes de programmation utilisant des aspects visuels spécifique sont en l'occurrences : les flux de données, les contraintes, la programmation par démonstration et à base de formes.

I.1.4.1.1 - Paradigme basé sur les flux de données

Ici, un programme est composé de modules fonctionnels avec des chemins entre les entrées et les sorties tel que les systèmes : LabView [VOS 86], CANTATA [YOU 95a], VIZ [HER 96]. Dans des langages textuels, les diagrammes de flux de données sont tracés comme les parties du procédé de représentation des programmes et sont alors traduits en texte ; les langages visuels omettent cette traduction.

I.1.4.1.2 - Paradigme basé sur les contraintes

Une contrainte peut affecter plusieurs variables qui à leur tour peuvent en affecter davantage. Cette forme d'interaction peut bénéficier des représentations schématiques. ThingLab [BOR 81] est un exemple d'un tel langage où un ensemble de contraintes (règles) décrit les propriétés invariables et les relations de tous les objets dans un espace de travail. Les solutions sont l'ensemble des valeurs qui satisfont à toutes les contraintes simultanément.

I.1.4.1.3 - La programmation par démonstration

La programmation est réalisée en manipulant graphiquement les données sur l'écran, montrant à l'ordinateur ce que le programme doit effectuer. Les exemples de ce type de programmation sont Thinker [LIE 87], Rehearsal World [GOU 84] (Influençant ultérieurement AUDITION), et PT (Pictorial Transformation) [AMB 93]. Ce dernier est un langage procédural utilisant la métaphore d'une fabrication de film (voir également Marquise [MYE 93] qui est un outil de Garnet [MYE 90b]).

I.1.4.1.4 - Paradigme à base de formes

Ce style de programmation peut être vu comme une généralisation de la programmation de tableur. Bien qu'il emploie la forme textuelle, les relations entre les données et les expressions de différents calculs sont représentées séparément de la forme elle-même, et ne sont pas décrites textuellement. De ce fait, le tableur est naturellement visuel. La représentation visuelle d'une matrice de cellule permet l'omission des concepts de variables, de déclarations, et de formatage des sorties. De plus, elle contribue à l'image visuelle d'une large matrice de cellules où chaque valeur est normalement évaluée juste une fois dans un ordre non spécifié formellement. D'autre part, les formes prolongent le paradigme du tableur. Le composant "feuille" de base est une forme sur laquelle les utilisateurs peuvent placer des cellules appelées objets. L'expression d'une cellule peut référencer une cellule (ou plusieurs) d'un objet sous la forme du contenu. Les sous formes peuvent être utilisées pour implémenter une certaine forme d'héritage.

I.2 - Problématique

Pour mieux comprendre ce que peut attendre un utilisateur de notre système et introduire la problématique de la visualisation graphique et plus particulièrement les problèmes liés à la visualisation des informations d'un algorithme nous allons prendre un exemple simple qui est un algorithme appelé tri par bulle.

Cet algorithme sert à mettre dans un ordre logique des données qui nous sont fournies ou acquises en désordre. Plus explicitement celui-ci consiste à trier une liste d'éléments par une séquence de comparaison entre deux éléments adjacents.

Pour améliorer la compréhension du principe utilisé par cet algorithme nous allons essayer de décrire de manière textuelle les règles opératoires qu'il emploie. Pour ce faire, nous

allons considérer un tableau contenant des éléments de valeur quelconque, que nous voulons remettre en ordre croissant.

A la première itération ou passe, l'algorithme consiste à comparer par adjacence les éléments du tableau du premier élément au dernier ; c'est-à-dire le 1er élément avec le 2ème puis le 2ème avec le 3ème , etc. Lors de cette comparaison, si le 1er élément est plus grand que le 2ème, nous effectuons un échange entre ces deux valeurs. Cet échange consiste à placer le plus petit des deux éléments à la première place et le plus grand à la seconde. A la comparaison suivante, et à l'échange éventuel qui s'en est suivi, nous avons le plus grand des trois premiers éléments, par rapport à l'état initial du tableau, en 3ème position. Mais il faut remarquer que le plus petit ne se trouve pas nécessairement à la première position. Ce qui nous suggère qu'il nous faudra nécessairement effectuer au moins une seconde passe pour trier ces deux éléments.

Si nous extrapolons maintenant cet algorithme pour *n* éléments, nous obtenons à la fin de la première passe le plus grand élément à la dernière position du tableau donc se trouvant à sa position triée ou finale. A la seconde passe, nous n'avons donc plus qu'à trier les *n - 1* premiers éléments et il nous faudra effectuer *n - 1* passes pour obtenir tous les éléments du tableau en ordre croissant.

Ainsi, après avoir déterminé les règles opératoires employées par l'algorithme nous pouvons implémenter celui-ci en écrivant un programme en langage Pascal (Cf. figure 1.1).

```
program triparbulle;
type a = array [1..50] of real;
var i, j, n, integer;
    temp : real;

begin
        writeln ("Entrez le nombre d'éléments du tableau");
        readln(n);

        writeln ("Entrez les éléments");
        for i := 1 to n do readln(a[i]);

        for j := n-1 downto 1 do
                for i := 1 to j do
                        if a[i] > a [i+1] then
                                begin
                                        temp := a[i];
                                        a[i] := a[i+1];
                                        a[i+1] := temp
                                end
end.
```

***Figure 1.1** : Un programme Pascal implémentant l'algorithme de tri par bulle.*

Le programme est constitué principalement de deux boucles. La boucle interne permet d'échanger les données lors de chacune des passes, et l'autre boucle contrôle les n-1 passes pour effectuer le tri complet du tableau. D'autre part, pour permuter les deux éléments, nous faisons appel à une variable temporaire servant de tampon d'échange.

Pour accompagner ce programme, une animation d'algorithme utilisant notre système peut lui être adjointe. L'animation peut débuter par la visualisation des éléments dans l'ordre initial dont la représentation pour cinq éléments se trouve en figure 1.2.

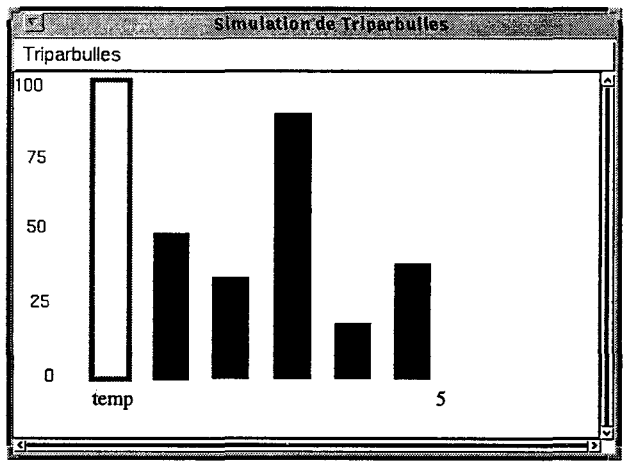


***Figure 1.2** : Première visualisation de l'animation d'un algorithme de tri par bulle.*

Les éléments sont représentés par des images rectangulaires dont les hauteurs sont proportionnelles à la valeur des données qu'elles représentent. D'autre part, l'affichage d'une échelle aide le visualisateur à évaluer, s'il le désire, la valeur des données contenues par le tableau. D'autres informations ont été rajoutées en l'occurrence le nombre d'éléments et l'état de la variable *temp*.

L'animation de l'algorithme de tri par bulle s'effectue lorsque l'accompagnement des actions de l'animation correspond à un événement ou un changement d'état du programme lors de son exécution. Ainsi, le changement d'état des valeurs *i* et *i + 1* fait modifier de couleur les images correspondant aux données pointées par ces index dans le tableau (dans l'exemple présenté, ci-dessus, *i* a pour valeur 1). Il est à noter que ces images retrouvent leur couleur d'origine quand elles ne sont plus pointées. L'échange des deux éléments, s'effectuant par l'intermédiaire de trois affectations et par l'emploi d'une variable temporaire nous avons choisi de visualiser ces trois événements et donc, par la même, la variable *temp*. Chacune de ces affectations correspondra à une translation entre la position de la donnée copiée ($2^{ème}$ valeur du tableau) et la position de la donnée affectée (la variable *temp* pour la figure 1.3). Par cette

visualisation, nous montrons qu'en Pascal une affectation correspond effectivement à une copie des données. Le mouvement de ces images est graduel et sans à-coup car nous utilisons plusieurs positions intermédiaires pour créer l'illusion d'un mouvement continu (les hachures derrière l'image rectangle ont été rajoutées pour donner l'impression de mouvement).

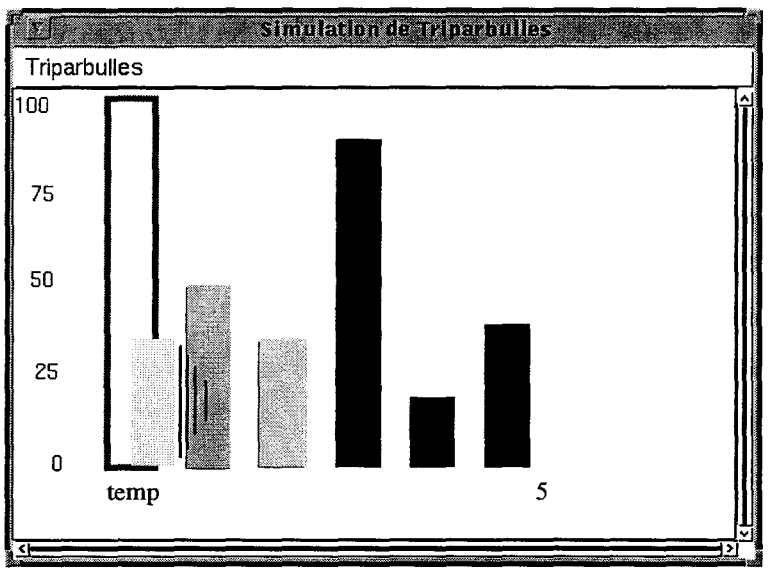


Figure 1.3 : *Animation de l'algorithme de tri par bulle.*

L'animation que nous venons de décrire n'est représentative que d'une partie des possibilités de notre système que ce soit pour l'animation des algorithmes ou pour la programmation visuelle dont nous n'avons pas du tout fait état. Cependant cet exemple nous permet de définir les principaux problèmes auxquels nous avons été confrontés et qui peuvent être extrapolés à la programmation visuelle étant également un système de visualisation graphique. Toutefois, d'autres problèmes plus spécifiques à ces deux outils peuvent être définis mais ne nécessitent pas d'exemple pour être présenté et compris.

I.2.1. Problèmes généraux.

Dans l'animation d'algorithme comme dans la programmation visuelle, une des premières tâches est de définir les aspects du programme que nous allons visualiser. Car formellement, un programme peut être caractérisé par son code, par ses états de contrôle, par l'état de ses données ainsi que par son comportement. Mais comme nous le verrons par la suite, les systèmes de visualisation se sont souvent limités à ne présenter qu'un sous-ensemble de ces aspects du programme.

Nous pouvons également nous demander quelle sorte d'information nous pourrons véhiculer par la visualisation. Le résultat d'une animation doit être raisonnablement informatif et cohérent par rapport aux données du programme. En fait, le problème concerne le niveau d'abstraction que l'on peut associer aux concepts présentés sous une forme graphique. Par exemple, la visualisation de l'expression en cours d'évaluation offre un très bas niveau de représentation de l'état de contrôle ; tandis que la visualisation de l'animation de l'algorithme de tri par bulle (cf. figure 1.3) cache les détails du code et dépeint le comportement du programme à un niveau plus abstrait. Pour la programmation visuelle le problème concernera principalement les représentations associées aux différents modules utilisés.

D'autre part, bien que la visualisation d'une centaine de lignes de texte n'est pas aisée, la taille des informations à afficher peut se révéler contraignante, que ce soit pour les programmes ou les données. Ainsi certaines représentations peuvent se révéler être plus importantes que le texte qu'elles remplacent. Par exemple, le problème se pose lorsque l'on a 1000 éléments d'un tableau à visualiser et non plus 5 ou 10. En outre, la programmation visuelle est obligée de se limiter, très souvent, à ne définir qu'une cinquantaine d'objets dans leur espace d'environnement [MCI 92].

La difficulté la plus fondamentale dans l'animation graphique est de déterminer comment nous allons créer une visualisation et par là même une animation. Ce problème concerne les moyens par lequel l'animateur précise les aspects du programme qui doivent être extraits et comment ils doivent être exposés ; c'est ce que nous avons appelé plus simplement dans l'exemple précédent la manière de déterminer les événements intéressants de l'algorithme. La résolution de ce problème est essentielle car la façon d'acquérir ces événements peut conditionner la flexibilité d'un système.

Enfin, la dernière grande question est de choisir un type d'interface, c'est-à-dire de déterminer les moyens à fournir aux différents utilisateurs pour visualiser ou créer les informations visuelles. Ce problème comprend deux catégories : d'une part, le choix d'une interface qui conditionne les mécanismes avec lesquels les utilisateurs peuvent interagir pour explorer et comprendre l'information qui est exposée, et d'autre part, l'aspect graphique, c'est-à-dire le vocabulaire graphique employé pour créer une visualisation. Ce vocabulaire se caractérise, plus explicitement, par les types d'objets graphiques mis à la disposition de l'utilisateur et les manières de combiner ces objets pour construire les images. Il inclut, également, les différentes définitions des événements visuels comme la création, la destruction, ou la modification d'un objet graphique. Ces définitions doivent, alors, être

suffisamment génériques pour que l'utilisateur n'ait pas à concevoir de nouvelles routines d'animation pour chaque configuration possible du programme.

Nous pouvons également soulever le problème de savoir quels types d'animations nous allons utiliser pour transporter l'information. Plus explicitement, il va falloir déterminer les moyens de transmission d'informations les plus judicieux à employer du type : auditif, graphique, etc. Concrètement, ce problème comprend les mécanismes heuristiques employés par l'utilisateur en construisant des visualisations et font appel très souvent à une combinaison de plusieurs sources de communication.

I.2.2 - Les problèmes spécifiques de la visualisation de programme.

Un des problèmes concerne la difficulté de définir de nouvelles présentations. Car, même si de récents systèmes fournissent des outils pour simplifier la conception de nouvelles images, il peut encore être très difficile de les concevoir sans aucune formation.

Enfin, en ce qui concerne la visualisation dynamique des données, il est très difficile de préciser quand la mise à jour de la représentation doit être effectivement effectuée.

I.2.3 - Les problèmes spécifiques de la programmation visuelle.

En ce qui concerne la programmation visuelle nous nous sommes placés dans un cadre particulier qui est la programmation à l'aide de composants logiciels que nous avons appelés modules et est basé sur une communication de flux de donnée.

Les problèmes que nous pouvons soulever sont, alors, spécifiquement liés à ce type de programmation, à savoir, définir la nature des modules employés, comment effectuer l'association de ces modules pour créer de nouveaux programmes, comment les représenter et définir un mode d'exécution.

I.3 - Contributions

Notre système a été conçu comme un environnement de programmation homogène composé d'un outil de visualisation de programme et un outil de programmation visuelle. Ceux-ci permettent respectivement d'obtenir sous forme graphique, l'animation des informations d'un algorithme et de créer un programme modulairement à partir de différents composants logiciels élémentaires.

L'originalité de l'outil de visualisation de programme, est de gérer l'espace de travail, en l'occurrence l'écran, par l'intermédiaire de fenêtres autonomes de visualisation d'informations dont la remise à jour de l'affichage est effectuée automatiquement et le plus indépendamment possible de l'algorithme considéré. Quant à l'outil de programmation visuelle, en dehors des concepts intrinsèques à ce type d'interface, son originalité est de faire appel à la visualisation de programme pour présenter le traitement des modules et des données employées sous forme graphique et animée.

Ainsi, le but de notre système est de pouvoir afficher, à partir de n'importe quels programmes ou modules, le comportement des informations qui les caractérise en donnant aux différents utilisateurs, si nécessaire, un apprentissage rudimentaire. Nous avons, pour ce fait, établi une interface conviviale s'appuyant sur des outils et des concepts nous permettant d'automatiser la construction et la gestion des différentes visualisations.

Une des grandes originalités de notre application réside par les interactions que l'utilisateur peut effectuer, au cours de l'exécution, avec ces deux types de simulation.

Ainsi, l'intérêt des chercheurs et des étudiants pour un tel outil se trouve accru car ils peuvent alors interagir pour comprendre, entre autres, comment leurs actions affectent les différentes simulations et par conséquent le système réel transposé sous forme algorithmique et informatique.

Ce type de simulation est appelé plus communément simulation visuelle interactive et a été introduite par Hurrion [HUR 78]. Elle implique que notre système autorise, par exemple, des modifications de l'algorithme au cours de son exécution, afin de visualiser immédiatement les effets induits de ces changements [BEL 87] (Cf. figure 1.4).

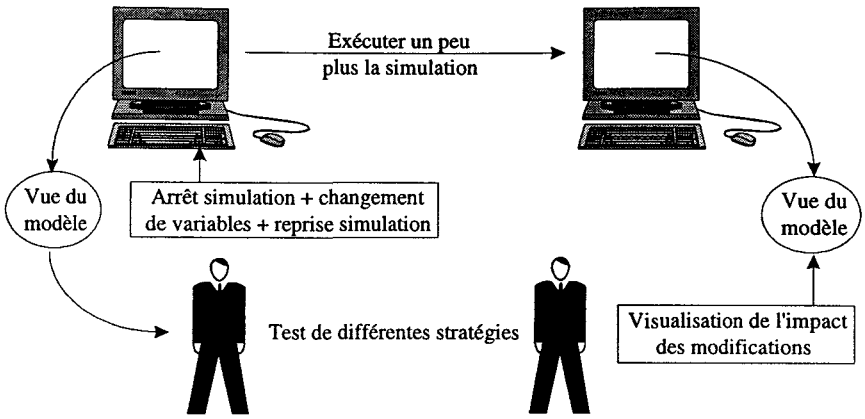


Figure 1.4 : *Exemple d'une séquence d'interaction.*

D'autre part, la plupart des travaux pour visualiser et animer des algorithmes ont fait ou font appels à trois participants : le programmeur qui développe le programme original, l'animateur qui définit et construit l'animation graphique, et le visualisateur qui observe la représentation graphique (Cf. figure 1.5).

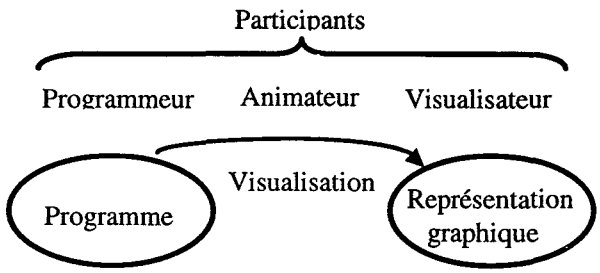


Figure 1.5 : *Modèle de visualisation de programme proposé par [ROM 93].*

Le visualisateur dans ce modèle ne peut alors créer de programme original ni d'animations graphiques.

Aussi, la mise en place d'une simulation conviviale et interactive et les facilités de mise en oeuvre automatique d'une animation ont pour objectif de minimiser le rôle des deux premiers participants, car une simulation interactive donne, intrinsèquement, un plus grand rôle au visualisateur en lui apportant la possibilité d'interagir avec la plupart de ces propres paramètres.

Notre système a donc été doté de plusieurs interactivités dont les plus importantes (en dehors des différents modes de simulation, des facilités de présentation, etc.) sont les suivantes :

- pouvoir changer la valeur des structures de données du programme,
- pouvoir changer le code source d'un algorithme ou d'un module,
- visualiser dans des fenêtres autonomes une expression, sélectionnée à la souris dans le texte source ou définie textuellement par l'utilisateur.

L'interactivité constitue alors un des critères de convivialité adopté lors de l'élaboration du système. Ces critères dépendent de l'ergonomie des interfaces homme-machine et sont :

- homogénéité du système qui repose sur le fait que les prises de décision, les choix de conception de l'interface (dénominations, formats, etc.) soient conservés pour des contextes identiques. Cette homogénéité s'obtient aussi bien structurellement que visuellement (toute information est visualisée dans une fenêtre autonome, toute représentation adopte la même architecture graphique, un algorithme peut être considéré comme un module, etc.).
- réduction de la charge de travail qui consiste à minimiser les actions de l'utilisateur pour obtenir l'accomplissement de certaines tâches ou la densité informationnelle (en automatisant la gestion des événements, la création d'une animation graphique, en éliminant de l'interface toute visualisation d'information obsolète, etc.).
- adapter le logiciel aux différents niveaux d'expérience des utilisateurs tout en permettant selon leurs stratégies ou habitudes de travail d'obtenir les informations qu'ils souhaitent et de réduire la phase d'apprentissage (en les laissant choisir les données qu'ils souhaitent visualiser, en construisant des bibliothèques d'algorithmes ou d'animation de base, en leur permettant d'enrichir leurs bibliothèques, en adoptant un formalisme de construction générique c'est-à-dire de même type, etc.).

- contrôler explicitement les actions de l'utilisateur tel que celui-ci doit toujours avoir la main et donc de pouvoir traiter le déroulement des traitements en cours et d'exécuter les opérations demandées, au moment demandé impliquant un dialogue homme-machine évolué dit interactif. Ce critère implique que le retour d'information ou feed-back doit être aussi immédiat que possible (le changement de valeur d'une donnée doit se répercuter au niveau de son affichage).

Ces critères et ce style de simulation nous a alors conduit à utiliser une approche orientée objet pour la conception de notre système, car cette forme de programmation est, à cause des concepts qui lui sont inhérents, l'un des meilleurs choix possibles pour obtenir un logiciel graphique homogène, interactif et évolutif. Nous nous sommes ainsi orienté vers l'utilisation du langage Smalltalk-80 pour implémenter notre système. Celui-ci a l'avantage de posséder une bibliothèque d'objets de base très exhaustive et est, de ce fait, perçu comme un très bon langage de prototypage.

CHAPITRE II

UN MODÈLE DE VISUALISATION D'ALGORITHMES ET SES COMPOSANTS

II.1 - UN HISTORIQUE DES LOGICIELS DE VISUALISATION

L'importance des représentations visuelles pour comprendre les programmes informatiques n'est pas nouvelle. Goldstein et Von Newmann [GOL 47] démontrent l'utilité des organigrammes, tandis que Haibt [HAI 59] développe un système pour les générer automatiquement de Fortran ou des langages de programmes assemblés. Knuth [KNU 63] développe un système similaire. Bien que ces expériences aient, très tôt, jeté un doute sur la valeur des organigrammes comme une aide à la compréhension des programmes [SHN 77], une étude comparative menée par Scanlan est plus encourageante [SCA 89].

Une approche différente est prise avec les films de Knowlton [KNO 66a, 66b], effectuée au laboratoire Bell, qui montre la manipulation de listes avec le langage L^6 (Laboratories Low-Level Linked List Language) [KNO 66c]. Ce travail a été le premier à utiliser des techniques de visualisation dynamique et à aborder la visualisation des structures de données. Le débogueur de Baeker [BAE 68] pour l'ordinateur TX-2 produit des images statiques de la visualisation de files, mais il est vivant et interactif. Baecker a continué ce

travail en adoptant une direction pédagogique qui l'a conduit à des systèmes qui montrent des structures de données abstraites lors d'exécution de programmes [BAE 75] et finalement aboutit au film "Sorting Out Sorting" [BAE 81].

Les années 70 voient un retour aux organigrammes avec le développement des diagrammes de Nassi-Shneiderman [NAS 73] à opposer à la nature non structurée des organigrammes standard. Roy et St Denis [ROY 76] développent alors un système pour générer automatiquement les diagrammes de Nassi-Shneiderman du code source par un procédé de compilation spécialisé.

Ce qui reste des années 70 est de "jolies impressions (pretty-printing)", un terme inventé par Ledgard [LED 75] pour décrire l'utilisation d'espacement, d'indentation, et d'agencements pour que le code source soit plus aisé à lire dans un langage structuré. Plusieurs systèmes pour de "jolies impressions" automatiques ont été développés, tels que NEATER2 [CON 70] pour PL/I et le système de Hueras et Ledgard [HUE 77] pour Pascal. Ces techniques s'avèrent relativement simples ; cependant, de récentes extensions à ce travail ont employé la composition informatisée et l'imprimante laser pour donner une meilleure présentation du code source. Ceci est allé d'un simple changement de style de la fonte des mots clés [BSD 88] à la visualisation de programmes avec SEE [BAE 90]. Ce dernier prend automatiquement un ensemble de programmes C et formate un "livre de programme" à partir d'eux. Le système WEB de Knuth [KNU 84] est similaire, mais combine la documentation et le programme dans un document utilisant un langage de programmation étendu.

Les années 80 voient le début de la recherche sur les logiciels de visualisation moderne avec l'introduction de la visualisation d'image bit-map et la technologie des interfaces fenêtrées. Le plus important système et le plus connu de ce domaine est BALSA [BRO 84] suivi de Balsa-II [BRO 88] qui a permis aux étudiants d'interagir avec des visualisations dynamiques de programmes Pascal. Beaucoup de prototypes et de systèmes de production utilisant les technologies modernes d'interfaces homme machine ont été développés depuis.

II.2 - MODELE D'UN LOGICIEL DE VISUALISATION D'ALGORITHMES

Toutes les parties d'un logiciel peuvent être modélisées comme une boîte noire qui, en fonction de données d'entrée particulières qui lui sont fournies produit des données de sortie. Celles-ci sont renvoyées à l'utilisateur, qui s'en sert pour définir le prochain ensemble d'entrées, comme schématisé à la figure 2.1.

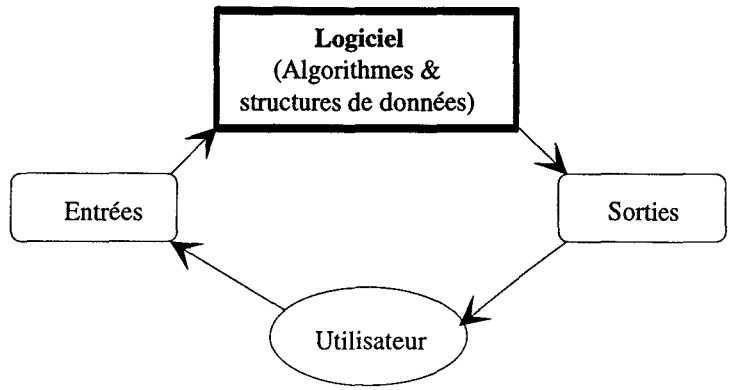


***Figure 2.1** : Modèle général d'un logiciel.*

Les données d'entrée peuvent être multimodales ; par exemple, elles peuvent être constituées de fichiers, d'ordres tapés à la machine, de gestes, de sons, etc. Et ces données peuvent être temporellement statiques ou dynamiques.

La boîte noire englobe un ensemble d'algorithmes arbitrairement complexes, qui pour des langages compilés sont représentés comme des instructions machine exécutées sur une architecture matériel spécifique. Conceptuellement, cependant, la boîte noire est constituée d'un haut niveau d'algorithmes et de structures de données originalement conçues par le programmeur.

Les données de sorties peuvent, également, être multimodales et temporellement dynamiques.

L'utilisateur peut interagir avec la donnée d'entrée ou de sortie, soit de manière peu fréquente, comme c'est le cas avec le traitement de données scientifiques, ou de manière continue, comme c'est le cas avec des applications hautement interactives de dessin.

Ce modèle simple mais complet, fournit une base conceptuelle pour définir une structure et déterminer les principaux composants d'un système de visualisation de programme.

En joignant l'interactivité à notre logiciel nous obtenons un modèle plus riche que celui de Roman [ROM 93] (Cf. figure 1.5) car ceci nous amène à introduire un nouveau participant que nous appelons "Utilisateur" qui est pour nous un visualisateur à la fois actif et passif. Celui-ci nous permet entre autre de boucler notre modèle et donc d'interagir avec le logiciel ce qui ne peut être systématiquement considéré pour toutes les phases d'utilisation pour le visualisateur défini par Roman qui est plutôt un utilisateur passif.

En fait on distingue en général trois types d'utilisateur dans un environnement interactif :

- l'utilisateur naïf qui interagit avec le système de manière occasionnelle et limitée. Il demande à être guidé par l'intermédiaire d'une interface intuitive et totalement auto-descriptive et utilise en premier lieu les bibliothèques et les exemples prédéfinis. Celui-ci, toutefois, ne reste pas forcément éternellement naïf.

- l'utilisateur intensif (professionnel) qui se base sur son expérience pour interagir avec le système pour contrôler le déroulement des traitements informatiques en cours, pour enrichir l'environnement de base, etc. Il faut donc pouvoir adapter le système selon ses besoins et ses préférences en ayant la possibilité de personnaliser l'interface.

- l'utilisateur occasionnel qui interagit faiblement avec le système s'apparente dans ce cas à l'utilisateur naïf. Ses attentes sont par contre différentes et demande surtout une interface cohérente et homogène lui permettant un apprentissage aisé et rapide (action identique pour modifier une information, présentation similaire de fenêtre présentant le même style d'information, etc.).

Le modèle présenté à la figure 2.2 montre que chaque intervenant (le programmeur, l'animateur, l'utilisateur ou le visualisateur et le développeur) contribue de manière distincte à l'animation.

Nous retrouvons dans ce modèle les trois participants identifiés par Roman qui n'interviennent que localement et de manière incomplète sur le modèle général du logiciel de visualisation d'algorithmes.

L'introduction du développeur du système nous permet alors de prendre en compte l'édification du système mais aussi la création de bibliothèques d'animations et d'algorithmes initiales. En ce qui concerne l'utilisateur, celui-ci nous permet d'introduire l'interactivité qu'il a avec le logiciel et suivant son expérience, les concepts de spécification utilisés et les outils fournis font transparaître qu'il peut remplacer le programmeur et/ou l'animateur.

Les structures du modèle font également apparaître les différents composants (nommés par les étiquettes verticales) constituant un logiciel de visualisation de programme.

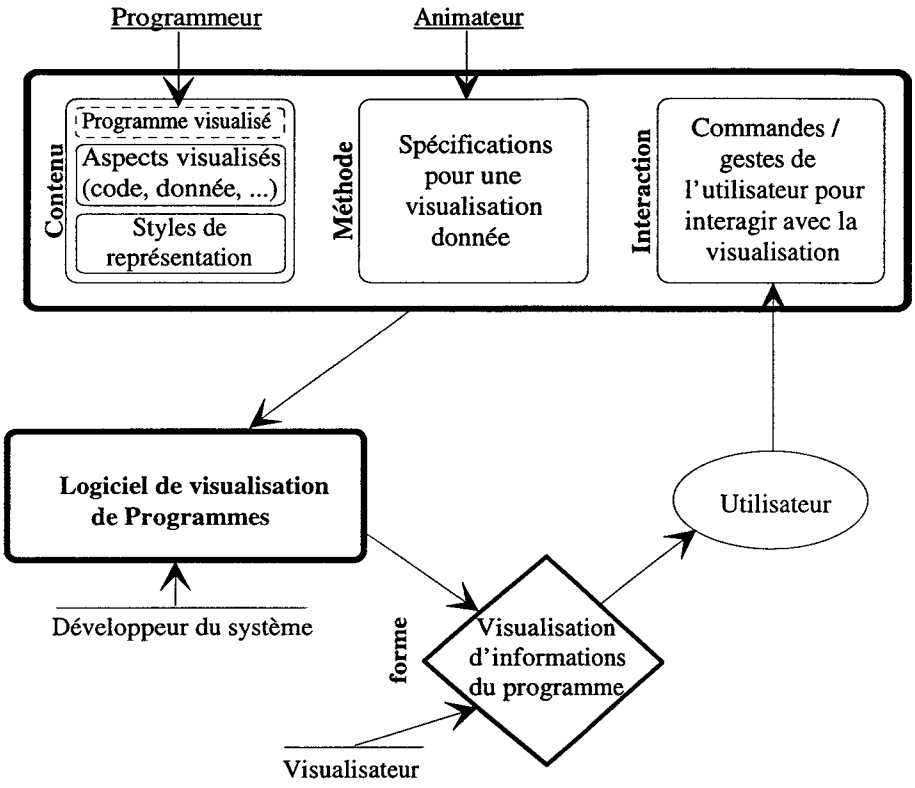


Figure 2.2 : Modèle général d'un logiciel de visualisation d'algorithmes.

- Le contenu regroupe, d'une part le programme que l'on veut animer et d'autre part les différents aspects du programme que l'utilisateur pourra visualiser. Ces aspects peuvent être dissociés en deux sous-groupes dont le premier concerne les différentes informations constituant un programme et le second la manière de représenter ces informations.

- La méthode caractérise les éléments importants de la spécification de la visualisation car elle détermine comment les informations vont être obtenues et transmises à la représentation. Elle établit le lien entre le programme et l'animation.

- La forme caractérise l'élément le plus important du point de vue de l'utilisateur car elle concerne la perception des informations mises à sa disposition. Elle regroupe le vocabulaire graphique employé, la gestion des éléments graphiques constituants de l'animation et enfin le type d'interface visuelle utilisée.

- L'interaction concerne les moyens mis à la disposition de l'utilisateur pour agir sur le logiciel à tous moments.

Nous allons pour chacune de ces catégories, dans la suite de ce chapitre, répertorier les techniques les plus typiques employées par les logiciels de visualisation ; ceci va nous permettre de définir les caractéristiques de notre propre logiciel.

II.3 - LE CONTENU

Cette catégorie détermine les types d'information qui peuvent être visualisés par un logiciel de visualisation d'algorithmes ainsi que le style de la représentation de ces informations pour tout programme que le programmeur ou l'utilisateur expérimenté à implémenté (Cf. figure 2.3).

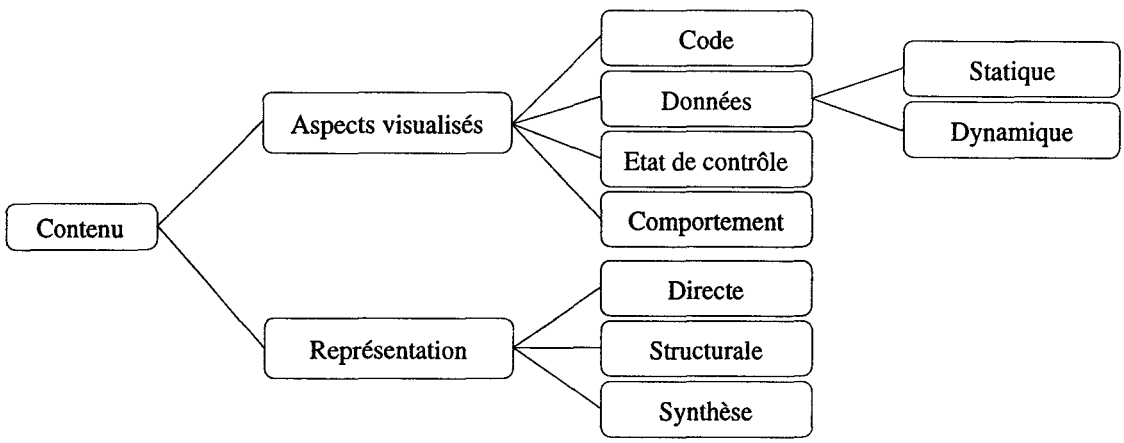


Figure 2.3 : *Sous catégories de la catégorie contenu.*

II.3.1 - Aspects du programme visualisé

Le but d'un système de visualisation de programmes est d'extraire l'information sur un certain aspect ou des aspects d'un programme et de présenter celle-ci sous une forme graphique. Ainsi, un programme peut être caractérisé par son code (par exemple, les expressions), l'état de ces données (par exemple, la valeur assignée à des variables) et par l'état de contrôle (par exemple, la procédure exécutée). Les propriétés dynamiques du programme sont capturées par "l'histoire de l'exécution", ou le "comportement", que nous définissons comme une séquence d'états dans lequel chaque paire d'états consécutifs est représentée par un événement atomique tel que l'exécution d'une expression. Il existe bien évidemment plusieurs manières de visualiser ces informations (Cf. figure 2.4).

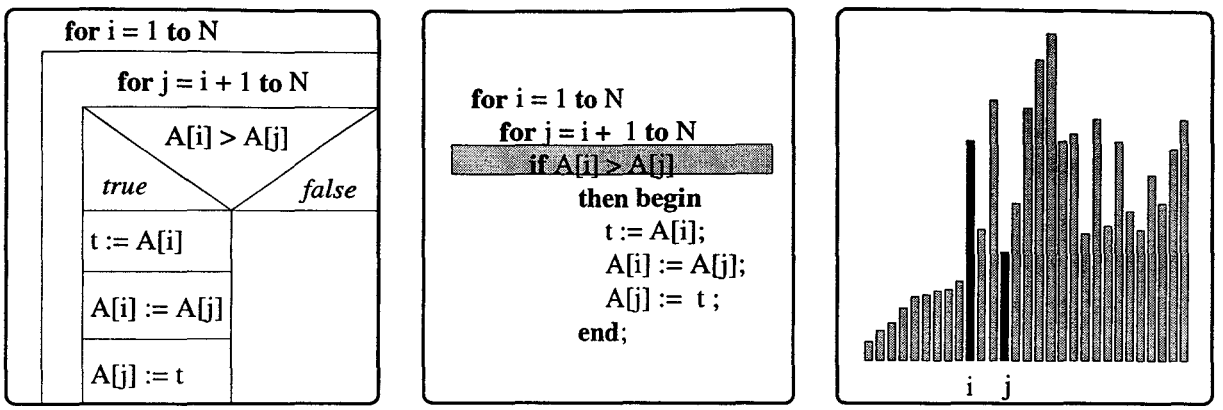


***Figure 2.4** : Exemple de visualisation d'informations : (a) le code ; (b) l'état de contrôle ; (c) l'état des données*

II.3.1.1 - Le code

La visualisation du code la plus simple est une présentation textuelle, plusieurs systèmes se sont toutefois attachés à le visualiser différemment. Ainsi la présentation du code peut être améliorée simplement en utilisant les diagrammes de Nassi-Shneiderman [ROY 76]. Une approche différente prise par le système SEE [BAE 90], vu précédemment, utilise les techniques typographiques pour visualiser les programmes C et permet par l'intermédiaire de ligne de commande de personnaliser ces visualisations pour un utilisateur donné. Le système Greenprint [BEP 80] quant à lui fournit un emboîtement de blocs et une représentation graphique d'un programme sous forme de boîtes qui sont placées à côté du programme textuel pour illustrer le contrôle de flux.

Mais la tendance actuelle va vers une représentation plus abstraite, ainsi il est possible de présenter un programme à partir des arbres syntaxiques en déclarant des schémas blocs d'interconnexion de modules de programme.

Dans Pegasys [MOR 85], les images sont des documentations formelles de programmes et sont dessinées par l'utilisateur et vérifiées par le système qui s'assure qu'elles sont syntaxiquement correctes et, dans une certaine mesure, si elles s'accordent avec le programme. Le programme peut tout de même être introduit dans un langage conventionnel (Pascal, Ada...).

Mais la plupart des systèmes, que ce soit d'animation d'algorithmes (par exemple Balsa [BRO 84], PV Prototype [BRO 85], MacGnome [CHA 85]) ou différents debogueurs, présentent le code sous une forme textuelle. De même, ils s'attachent, non pas à animer le code lui même, mais montrent dynamiquement, ligne par ligne, la partie du code évaluée lors de l'exécution du programme.

II.3.1.2 - Les données

Se plaçant au-delà des vues purement textuelles, quelques systèmes fournissent des vues graphiques sur les données et les structures de données d'un programme. Ces systèmes désirent donner aux utilisateurs la possibilité d'inspecter les valeurs des variables et d'identifier plus facilement les erreurs qu'en employant des méthodes traditionnelles. Nous les avons classés en deux catégories suivant qu'ils visualisent les données de manière statique ou dynamique.

II.3.1.2.1 - Visualisation statique

Le premier système de visualisation statique des données est le debogueur de Baeker [BAE 68], limité à la visualisation de file. Incense [MYE 83] génère automatiquement des visualisations d'images statiques pour les structures de données de tout type. Il permet à l'utilisateur de spécifier un format d'affichage particulier ou d'utiliser un format par défaut. Ces images comportent des lignes courbes et des flèches pour les pointeurs et des empilements de boîtes pour les tableaux et les structures de données. Le système GDBX de Baskerville [BAS 85], extension graphique du debogueur DBX, permet également d'afficher les données sous une forme graphique. Il permet de plus différentes formes d'interactions dont la principale est de pouvoir changer la valeur d'une donnée dans le programme en changeant la valeur inspectée. Le système Provide [MOH 88] présente les mêmes facilités mais il est basé sur un enregistrement de tous les changements d'état d'un programme (par exemple, le changement de la valeur d'une variable) et peut donc fournir un historique de son exécution.

II.3.1.2.2 - Visualisation dynamique

Pour la visualisation dynamique des données, le premier système est celui de Knolton [KNO 66c]. Myers et ses collègues à Garnegie Melton ont développé un système automatique de visualisation de données appelé Améthyste [MYE 88]. Celui-ci est intégré à l'environnement de programmation Pascal MacGnome [CHA 85] pour les ordinateurs Macintosh. Ce système est maintenant connu sous le nom de Pascal Génie [CHA 91], logiciel commercialisé. Plusieurs idées fondamentales d'Incense ont été incorporées au système Améthyste, qui peut présenter par défaut des représentations statique et dynamique des structures de données de programmes écrits en Pascal. Il suffit de pointer sur une variable avec la souris, et une image de cette donnée est automatiquement affichée. Le système Améthyste diffère d'Incense dans le fait qu'il est spécifiquement conçu pour être utilisé par des étudiants et est bâti dans un environnement intégré qui comprend des facilités pour diriger automatiquement la visualisation et l'animation.

Des évaluations empiriques suggèrent que l'utilisation d'un environnement du même style que Améthyste ou d'autres systèmes est plus efficace qu'une édition conventionnelle d'un programme [GOL 89] car il permet de mieux aborder les concepts d'un algorithme en visualisant dynamiquement et graphiquement son comportement et celui de ses données.

II.3.1.3 - L'état de contrôle

L'état du processeur de programme représente l'état de contrôle d'un programme séquentiel, mais très peu de systèmes emploient un si bas niveau d'abstraction. Ainsi, les informations les plus couramment visualisées concernent, la séquence des procédures invoquées et la ligne ou l'expression évaluée. Beaucoup de systèmes d'animation d'algorithmes fournissent un tel mécanisme mais représenté de manière différente.

Par exemple, une approche employée par le système Pecan [REI 85] est de rattacher le contrôle d'information à la structure générale du programme. Ceci est réalisé en visualisant la structure de code comme un diagramme d'interconnexion de module et en soulignant le module ayant actuellement le contrôle. Pecan contient un large ensemble de visualisations y compris la table des symboles, l'état de la pile, le type des données, le programme sous forme d'organigrammes et la visualisation d'expression sous forme d'arbre syntaxique.

II.3.1.4 - Le comportement

Nous pouvons voir un programme comme exécutant une série de transformations atomiques de son état. En observant ces transformations, reconnues formellement comme des événements, nous pouvons développer une compréhension d'un algorithme en visualisant son comportement c'est-à-dire la manière de transmettre le sens et sa méthodologie de travail. La diversité des événements peut varier d'une instruction machine à de larges exécutions d'opérations en considérant des blocs entiers de traitement. Les événements considérés comme intéressants peuvent intégrer des changements de valeur de variables spécifiques, des entrées et des sorties de procédures, ou des activités de communication. Ce type de visualisation peut consister, également, à visualiser l'évolution d'une ou plusieurs informations dans un certain contexte, c'est-à-dire son comportement vis à vis d'autres informations.

Un système de visualisation de programme obtient les événements en contrôlant une évaluation. Il peut exposer les résultats de manière "vivante" ou sauvegarder une trace de l'exécution pour la visualiser ultérieurement, ce qui permet au visualisateur d'aller et venir à différente vitesse dans le cheminement de la visualisation. Mais, nous reviendrons sur les différentes façons d'obtenir les événements dans la méthode de spécification (Cf. chapitre II.4).

II.3.1.5 - Le langage utilisé et les aspects visualisés

Les aspects d'un programme qui peuvent être visualisés dépendent des paradigmes de programmation sous-jacents au langage utilisé (Cf. tableau 2.1). Les aspects, qui sont répertoriés ci-dessus, correspondent naturellement à une certaine classe de langages. Pour des langages fonctionnels, d'autres aspects d'un programme sont importants.

Systèmes	Kaestle et Footscape	TPM	Object Oriented diagramming	Pavane	MRE	Polka
Types de langage visualisé	Lisp	Prolog	Langages orienté-objet	Langage parallèle, Swarm, ou C.	Langage parallèle PARLOG	Langage parallèle basé sur C ou C++

***Tableau 2.1** : Aspects visualisés particuliers pour quelques systèmes*

Le système Kaestle et Fooscape [BOC 86] présentent des structures de liste et le réseau d'appel de fonction à l'intérieur d'un programme LISP. Lieberman [LIE 89] utilise une unique représentation à trois dimensions pour visualiser les structures du programme LISP. Dans un environnement de programmation logique, les clauses pertinentes et les buts sont les aspects critiques à visualiser. Par exemple le système TPM (Transparent Prolog Machine) [EIS 88, BRA 91b] est un interpréteur qui visuellement dépeint les traces de programmes Prolog tandis que London et Duisberg [LON 85] ont été les premiers à travailler avec Smalltalk.

Quant au système "Object-Oriented Diagramming" [CUN 86], il a un centre d'intérêt différent. Il vise à présenter le passage de message dans des programmes orientés objet. Le système dessine les objets comme des boîtes ; des flèches montrent si le message est passé par l'objet lui même ou par une de ces super-classes.

Une autre forme de visualisation d'algorithmes concerne les algorithmes parallèles. Dans ce domaine la collecte des données est plus automatique car elle est souvent basée sur l'échange de messages ou sur l'état des processeurs. Seul Pavane [ROM 92b] est vraiment un système conçu pour fournir des visualisations simultanées d'éléments d'un programme, écrit en Swarm [ROM 90], avec de récentes extensions permettant la visualisation de programmes C. Price et Baeker [PRI 90, 91] ont également construit un prototype pour un langage procédural qui montre l'activité des processus dans une représentation statique de module et de procédures hiérarchiques ainsi que des vues individuelles montrant le statut de chaque processus. Le système MRE de Brayslaw [BRA 90, 91a] peut visualiser le langage logique PARLOG, une version parallèle de Prolog. Le système Polka [STA 92], qui est le

successeur de Tango, est également adapté pour la visualisation d'algorithmes parallèles. Pour un aperçu plus détaillé des systèmes de visualisation d'algorithmes parallèles, le lecteur est invité à se référer à l'étude de Kraemer et Stasko [KRA 93].

Si la plupart des systèmes de visualisation de programmes travaillent avec des langages plus traditionnels que TPM et Pavane pour ne citer qu'eux, le système ANIM [BEN 91, 92] est remarquable, dans ce cas particulier, par sa capacité à travailler avec énormément de langage dans le sens ou ANIM est un langage indépendant de l'algorithme à animer.

Si la plupart des systèmes sont techniquement capables de fournir des visualisations de tous les algorithmes employant un certain langage, certains se sont spécialisés dans des visualisations particulières. Par exemple "Sorting Out Sorting" est spécialisé dans neuf algorithmes spécifiques tandis que BALSA montre des visualisations de traitement de tableau et de parcours de graphe. Pascal Génie et ObjectCenter [CEN 91], qui est une extension des fonctionnalités du debogueur "DBX" d'Unix, emploie des graphiques pour transmettre certaines informations telles que des listes chaînées et des arbres. Quant à UWPI (University of Washington Program Illustrator) [HEN 90a, 90b] il peut seulement visualiser des structures de données simples pour un petit ensemble de recherche dans des graphes et d'algorithmes de tri de tableau.

II.3.2 - La représentation

Alors que différents systèmes de visualisation de programme peuvent visualiser les mêmes aspects d'un programme, le niveau d'abstraction présenté par chacun d'eux peut varier radicalement. Par exemple, une visualisation d'une structure de donnée d'un système peut exprimer trois noms de variables entières heures, minutes et secondes comme trois boîtes rectangulaires contenant les valeurs sous forme textuelle. Cependant, une autre représentation de cette structure de donnée peut être obtenue sous la forme d'un cadran avec le positionnement approprié des aiguilles représentant les heures, les minutes et les secondes (Cf. figure 2.5).

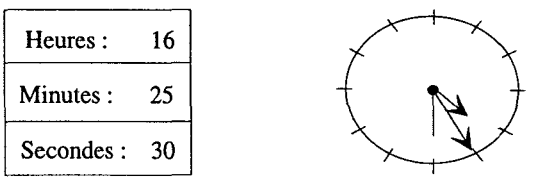


***Figure 2.5 :** Deux formes de visualisation d'une structure de donnée.*

Roman et Kenneth [ROM 93] ont distingué trois niveaux de représentation qui sont une classification du niveau d'abstraction de l'information visualisable. Les frontières entre ces différents niveaux sont imprécises, et en pratique, dans un système de visualisation il est plausible de supporter plusieurs niveaux d'abstraction, séparément ou en les combinant.

II.3.2.1 - La représentation directe

Selon Roman et Kenneth, le plus bas niveau d'abstraction pour une représentation graphique est l'établissement d'une relation directe entre l'aspect d'un programme et une image. Dans ce cas, les mécanismes d'abstraction sont si limités qu'il est possible de faire le chemin inverse c'est-à-dire reconstruire aisément l'information originelle à partir de la représentation graphique.

II.3.2.2 - La représentation structurale

Nous pouvons obtenir des représentations plus abstraites en concentrant l'attention du visualisateur sur des aspects particuliers du programme ou de son exécution. Une des façons de procéder est de cacher ou d'encapsuler l'information hors de propos et d'effectuer une représentation directe de l'information restante. Une autre manière de procéder est d'accentuer des portions importantes de l'information dans l'image finale par différentes techniques.

II.3.2.3 - La représentation de synthèse

Les représentations de synthèse sont distinctes des représentations structurales dans le fait que l'information intéressante n'est pas directement représentée dans le programme mais est dérivée des données du programme. Ce changement de perspective survient souvent quand les représentations de données sélectionnées par le programmeur rentrent en conflit avec les besoins de l'animateur. L'animateur peut préférer accentuer d'autres aspects de l'algorithme, qui, bien que logiquement présents dans le programme, n'ont pas de représentation explicite. L'animateur doit alors construire et entretenir une représentation qui est plus commode pour la visualisation. Beaucoup de systèmes supportent des formes de représentations de synthèse. Dans Tango [STA 89, 90], par exemple, l'animateur peut construire des représentations de synthèse en définissant des structures de données partagées par les procédures graphiques.

II.4 - LA METHODE DE SPECIFICATION

Cette catégorie décrit les caractéristiques fondamentales d'un logiciel de visualisation pour créer une visualisation (Cf. figure 2.6). Elle comporte la manière d'obtenir les données à la compilation ou à l'exécution ou aux deux, et les différentes techniques employées pour lier une visualisation au code source du programme.

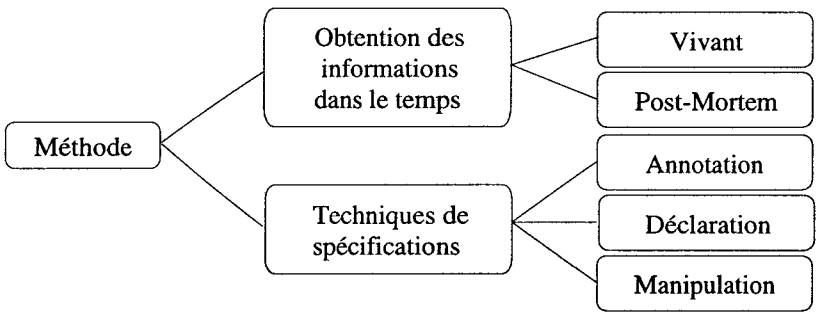


Figure 2.6 : *Sous catégories de la catégorie méthode.*

II.4.1 - Obtention des Informations dans le temps

Les informations d'une visualisation dépendent des données du programme recueillies à la compilation, à l'exécution ou aux deux.

En général les systèmes dont les informations dépendent des données assemblées uniquement à la compilation (tel que SEE) sont limités à visualiser le code du programme et les structures de données. Ces systèmes ne peuvent produire une quelconque visualisation des valeurs réelles des données, comme ils n'ont pas accès à cette information.

La génération des systèmes de visualisation de données, obtenues à l'exécution, peuvent produire des visualisations complexes de la variable d'espace utilisée par le programme et compte souvent sur l'animation pour effectuer une relation intuitive entre les aspects temporels du programme exécuté et la présentation de la visualisation.

UWPI, Pascal Génie et ObjectCenter recueillent leurs informations à la compilation et à l'exécution pour produire leurs visualisations car ces systèmes montrent des informations statiques du code et des informations dynamiques sur les données du programme.

Nous pouvons également subdiviser en deux catégories la relation temporelle appliquée à la visualisation et qui est basée sur l'obtention des données au cours de l'exécution. Nous trouvons ainsi des visualisations dont le style est dit vivant par opposition au style dit post-mortem.

Le style post-mortem (employé par ANIM et TPM) consiste à produire une visualisation en différé. Pour ce faire, nous enregistrons les informations sur la ou les données que nous voulons visualiser au cours d'une exécution, créant ainsi une base de données sur leurs états successifs. Ce style de visualisation a l'avantage de rendre disponible une information très riche avec l'occasion d'en disposer de manière optimale, que ce soit sur un état passé ou futur. Toutefois l'utilisateur est incapable d'interagir avec la visualisation, car il ne peut voir les effets de son interaction immédiatement.

Le style vivant consiste à recueillir et prendre en compte les informations au cours de l'exécution du programme. Cette méthode est la plus intéressante, bien que TPM permette les deux modes. L'avantage de ce style est bien entendu d'autoriser l'utilisateur à interagir avec l'ensemble des données au cours de l'exécution du programme.

II.4.2 - Techniques de Spécification

La méthode de spécification renferme les moyens par lesquels l'animateur peut préciser comment doivent être extraits et exposés les aspects d'un programme. Nous avons répertorié dans la littérature trois manières principales de définir et contrôler les événements dans un système de visualisation de programme (annotation, déclaration, manipulation ou démonstration) [MYE 90a, ROM 93, PRI 93].

II.4.2.1 - Par Annotation

Cette approche a été utilisée initialement par le système BALSA [BRO 84] puis par bien d'autres systèmes d'animation d'algorithmes (ses successeurs BALSA II [BRO 88] et ZEUS [BRO 91a] ainsi que les systèmes TANGO [STA 89, 90] et WinSanal [SZM 97] pour ne citer que les principaux).

Le principe consiste à écrire des procédures qui vont construire et modifier des images, puis augmenter le code source du programme en introduisant des appels à ces procédures. Le placement de ces appels (annotations) permet d'obtenir des "événements intéressants" à des moments estimés significatifs de l'exécution, pour ainsi obtenir des informations concernant les modifications subies par les structures de données sous-jacentes (Cf. figure 2.7). Ainsi, l'information sur l'état du programme est passée par les paramètres des procédures d'animation invoquées. A chaque figure est associé un programme appelé "vue", qui réagit aux événements intéressants et produit des événements visuels qui correspondent à des changements de la représentation graphique. Dans les algorithmes, les appels de

procédures servent à signaler les événements intéressants. Ces insertions de code n'apparaissent pas dans le texte visualisé car elles sont filtrées à la phase de formatage.

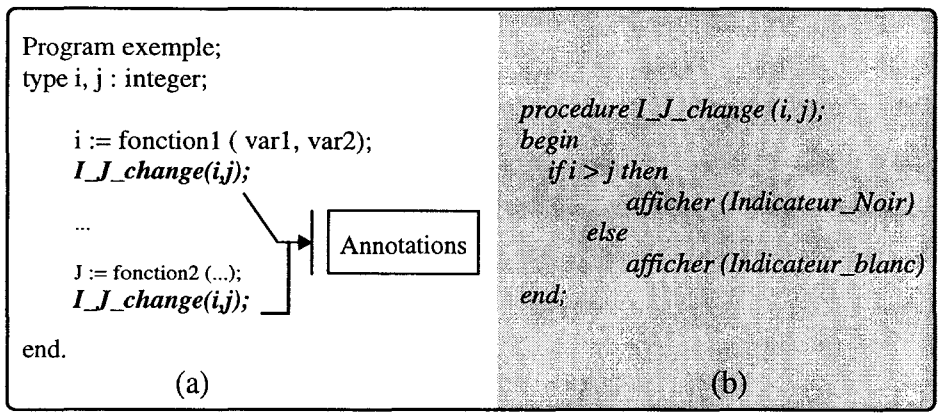


***Figure 2.7** : Spécification par annotation : (a) programme Pascal annoté; (b) code d'affichage*

Le système TANGO utilise une approche similaire mais facilite la production d'animations dans lesquelles les localisations d'objets et autres attributs peuvent être changés facilement, suivant des trajectoires spécifiques. De même, l'introduction de l'environnement Field [REI 90] permet au visualisateur d'annoter un programme (utilisateur) en utilisant un éditeur spécial ; si bien que le code source original reste visuellement inchangé (bien que le code exécutable diffère de la version non annotée). Les visualisations de TANGO sont traitées comme une relation entre les événements du programme et les actions d'animation. L'existence d'un événement d'intérêt peut déclencher, par exemple, des mouvements coordonnés d'un ou plusieurs objets graphiques dans l'image.

L'approche annotative a plusieurs avantages, dont la principale est qu'elle donne à l'animateur le choix de définir à quel niveau les événements sont déclenchables. Par exemple, un algorithme de tri peut être animé en exposant chaque comparaison et échange ; cependant, l'animateur pourrait choisir de ne présenter que le résultat final. D'autre part, la possibilité d'écrire dans l'application des procédures d'animation spécifique pour manier chaque événement est à la fois un avantage et un inconvénient, elle permet d'être adaptable mais requiert davantage de travail. L'utilisation de bibliothèques de routines peut quelque peu réduire ce travail. Mais, le défaut le plus significatif de la méthode est le besoin d'accéder et de modifier le code du programme. Un changement de code programme implique une redéfinition de l'animation.

II.4.2.2 - Par Déclaration

Dans l'approche déclarative, l'animateur précise une relation entre l'état du programme et l'image finale, et les changements d'état sont aussitôt répercutés dans l'image. Les déclarations de types simples laissent l'animateur mettre directement en relation les valeurs des variables de l'algorithme avec les différents attributs des objets graphiques de l'image finale. Cette approche (dans laquelle la déclaration est limitée à gérer des relations) pourrait être appelée associative, étant donné qu'elle demande à l'animateur qu'il définisse seulement une à une des associations entre variables et valeurs.

L'approche déclarative plus généralement permet de définir des relations avec toutes les formes de représentation, d'obtenir et de visualiser des expressions ou valeurs d'expressions ainsi que les attributs de l'état du programme. G. C. Roman a été très tôt intéressé par cette approche [ROM 89] et l'a amené à développer un système de visualisation d'algorithme appelé Pavane [ROM 92b] qui est le premier système à employer des relations déclaratives de manière très générale.

La figure 2.8 illustre l'approche déclarative d'une spécification utilisant une notation à base de règle comparable à ce qui peut être défini dans Pavane. Explicitement la modification du code du programme n'est pas nécessaire, mais le système doit fournir des mécanismes (opérant lors de la compilation ou de l'exécution) pour extraire les informations dont il a besoin.

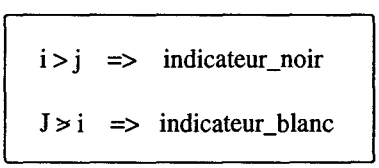


Figure 2.8 : Spécification déclarative

La visualisation découplée des événements dans le traitement sous-jacent a de nombreux avantages. En particulier, les traitements dans lesquels les événements sont difficiles à définir ou à isoler peuvent être plus facilement visualisés en utilisant la déclaration plutôt que l'annotation, et les spécifications sont assez souvent compactes car elles se réduisent généralement à deux ou trois conditions.

Cependant, la déclaration a des inconvénients. Un problème survient typiquement quand plusieurs changements d'état primitif doivent être considérés comme un seul changement logique et que l'image ne doit pas, par conséquent, montrer les états intermédiaires. Les systèmes peuvent, toutefois, éviter ce problème de plusieurs manières. Dans Alladin [HEL 89], par exemple, les relations entre les variables du programme et les

objets graphiques sont définies déclarativement, tandis que les annotations du code indiquent où l'image doit être mise à jour. Quant au système Animus [DUI 87], il emploie l'approche déclarative de deux manières. Chaque objet peut avoir une représentation graphique qui est automatiquement mise à jour en réponse aux changements de l'objet. Plus significativement, l'animateur peut déclarativement spécifier des contraintes sur les relations entre objets, et le système doit s'assurer que ces contraintes sont satisfaites (par exemple, en bougeant la représentation d'un objet).

II.4.2.3 - Par Manipulation

Les systèmes qui utilisent la manipulation, aussi appelée "animation par démonstration", spécifient les visualisations par l'emploi d'exemples [CYP 93]. Le premier système employant cette méthode pour l'animation d'algorithmes a été Gesture [DUI 87]. En fait, ces systèmes essaient de capturer les gestes utilisés par l'animateur en manipulant les objets dans une image et en attachant ces gestes à des événements spécifiques du programme ; cela a pour effet de définir une relation entre le programme et les événements visuels. Par exemple, pour animer des échanges d'éléments d'une collection dans un algorithme de tri, l'animateur pourrait définir un "geste d'échange" qui échange les positions de deux rectangles, l'animateur attacherait alors ce geste à l'événement d'échange dans l'algorithme en sélectionnant la portion appropriée du code dans le programme.

Mais, il est difficile de spécifier l'exact rapport entre le geste et les données ou les événements du programme. Ce rapport peut être précisé pour des cas spécifiques, mais une approche générale doit encore être définie. La plupart des systèmes existants emploient une approche mixte, par laquelle, les mouvements sont spécifiés gestuellement, mais la liaison entre les événements et certains attributs d'un mouvement est précisée par annotation ou déclaration. Le système Tango [STA 91] a employé cette approche expérimentalement.

II.4.2.4 - Autres techniques

Une autre technique utilisée par UWPI consiste à générer automatiquement une abstraction visuelle d'une donnée à partir de représentations prédéfinies. Le coeur de ce système est sa technique "d'inférence (ou de déduction)" qui analyse chacune des structures de données du code source et suggère un nombre d'abstractions plausibles pour chacune de ces structures. A celles-ci, était attribué un poids basé sur la proximité d'accès entre les opérations permises sur l'abstraction et les opérations trouvées dans le code. L'abstraction avec le poids le plus élevé est choisie pour chacune des structures de données du code et les résultats sont transmis à "l'agenceur de stratégie". Cet agenceur, connaissant la représentation graphique à utiliser pour chacune des abstractions de données, choisit la plus grande et la plus complexe

pour l'utiliser en tâche de fond tandis que le programme est exécuté. Le défaut principal d'un système automatique comme UWPI est qu'il peut produire une abstraction trompeuse pour une structure de donnée particulière.

Il existe encore d'autres techniques, dont l'emploi est limité. Par exemple, Zimmerman [ZIM 88] ainsi que le système de Flinn et Cowan [FLI 90] utilisent un moniteur qui "écoute" le bus de données ou une autre partie du matériel et montre un signal de commande vivant c'est-à-dire lors de l'exécution du programme.

II.5 - FORME DE L'INTERFACE

Nous allons maintenant nous intéresser aux caractéristiques de sortie du système, qui est en l'occurrence la visualisation perçue par l'utilisateur, répertoriées et hiérarchisées dans la catégorie forme (Cf. figure 2.9). Celle-ci dépend d'une part du type d'image utilisé ainsi que du vocabulaire graphique associé et de la gestion de ces images dans la simulation.

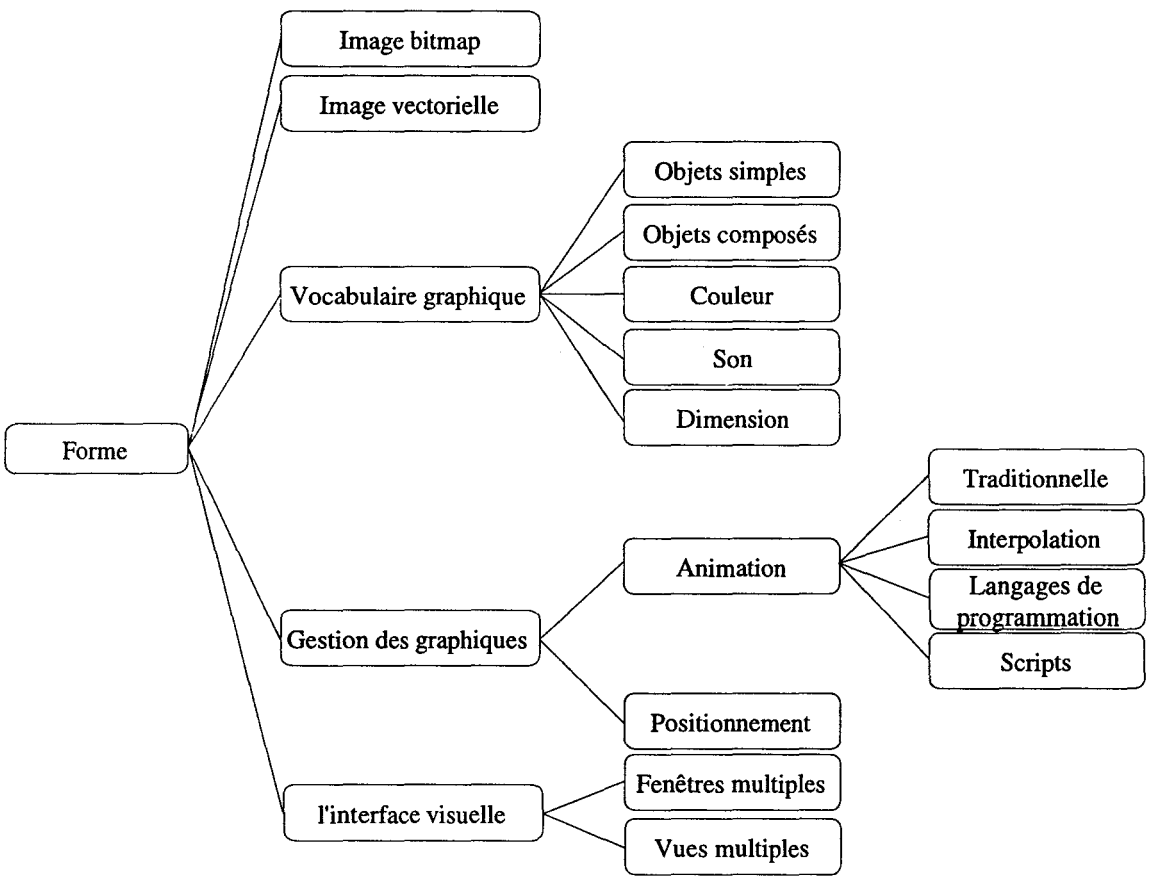


Figure 2.9 : *Sous catégories de la catégorie forme.*

II.5.1 - Image bitmap ou image numérique

Un bitmap est une structure de donnée contenant des informations qui décrivent une image rectangulaire. Celui-ci est constitué d'un certain nombre de lignes dont chacune définit des séries de points image ou pixels (Cf. figure 2.10).

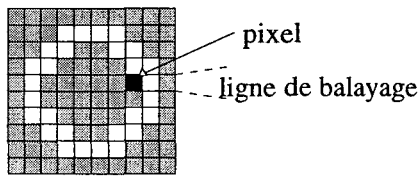


***Figure 2.10** : Eléments composants une image bitmap*

A chaque point correspond une valeur de couleur stockée sur un certain nombre de bits. Par exemple, pour une image définie avec une palette de seize couleurs, chaque point est stocké sur quatre bits dont les trois premiers correspondent aux valeurs spectrales, rouge, vert, bleu et le quatrième contrôle l'intensité. Le désavantage d'une telle représentation vient de la gestion de toutes les informations que constituent les points de l'image. Par exemple, pour une image 100*100 il faut gérer au moins 10000 données.

Ce type de représentation dite photographique correspond au codage le plus "primitif" de l'image et son emploi n'est motivé que si nous devons gérer point par point celle-ci. Ceci ne se justifie généralement pas dans l'animation d'algorithme car nous ne faisons appel qu'à des images symboliques qui utilisent plus particulièrement des diagrammes, des graphiques, des schémas synoptiques pour exprimer des informations quantitatives, topologiques, structurelles, etc.

II.5.2 - Image vectorielle

Dans le mode vectoriel, chaque objet graphique est créé à partir d'une définition géométrique des formes (ligne, cercle, courbes de Bézier, etc.) auxquelles sont associés des attributs (couleur épaisseur, etc.).

Chaque objet est stocké, dans ce cas, non sous sa forme de points en mémoire, mais sous la forme de primitives géométriques dont on ne conserve que les éléments significatifs, tels que le centre et le rayon d'un cercle, les courbures ou les longueurs de segments, ..., qui permettent à tout moment de retracer l'image sans altération.

Les principales caractéristiques d'une telle représentation sont les suivantes :

- La description vectorielle permet de hiérarchiser les différents objets graphiques, et par conséquent de créer des images complexes à partir d'éléments plus simples.

- Les objets graphiques peuvent facilement être modifiés de manière interactive sans altérer le reste de l'image ; il est ainsi très aisé à tout moment de déplacer, de superposer, de copier, de faire tourner, ..., les éléments souhaités.

II.5.3 - Vocabulaire graphique

Le vocabulaire d'un système graphique détermine les types d'objets graphiques que l'animateur peut employer et les manières de les combiner pour construire des images. Les principaux composants d'un vocabulaire graphique sont décrits ci-dessous.

II.5.3.1 - Les objets simples

Le plus simple type est une entité géométrique abstraite telle qu'un point, une ligne, un rectangle, un cercle ou une sphère. Nous pouvons inclure des splines, des icônes et du texte alphanumérique dans cette catégorie (Cf. figure 2.11).

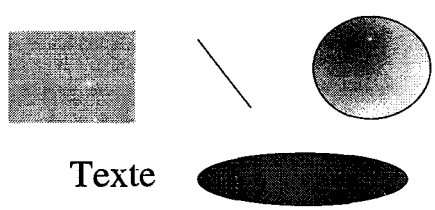


***Figure 2.11 :** Exemples d'objets simples*

Virtuellement, tous les systèmes de visualisation de programmes fournissent un large vocabulaire d'objets géométriques. En pratique, ces objets sont traités comme des classes d'objets que l'animateur instancie en fournissant des valeurs aux attributs des objets (pour une ligne ses points extrêmes et sa couleur ou pour une sphère son centre, son rayon et sa couleur).

II.5.3.2 - Les objets composés

Beaucoup de systèmes procurent une manière de créer de nouveaux types d'objets géométriques à partir d'une collection de simples objets (Cf. figure 2.12).

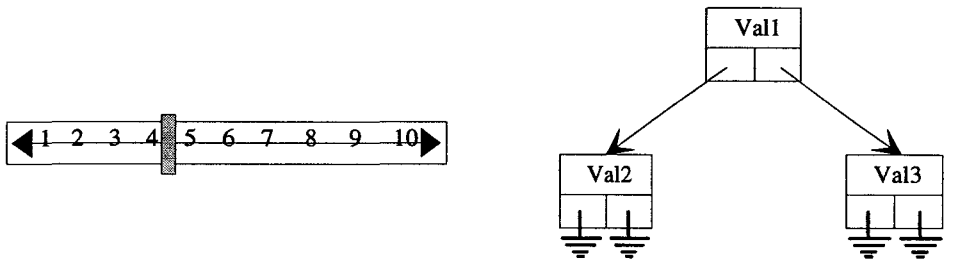


Figure 2.12 : Exemples d'objets composés

L'utilisateur définit les types d'objets qui peuvent être utilisés exactement comme s'ils étaient des types de base du système. L'objet composé créé possède des attributs hérités des objets simples et des attributs dérivés pour les nouveaux paramètre provenant de la composition (liste des objets simples, position relative, etc.). Tango est caractérisé par un tel mécanisme de définition de type et beaucoup de systèmes fournissent une variété d'objets composés prédéfinis, qui sont souvent appelés "widgets".

II.5.3.3 - La couleur

Certains attributs graphiques comme la couleur peuvent transmettre énormément d'information s'ils sont judicieusement utilisés. Tufte [TUF 83, 90], donne d'excellents conseils sur l'utilisation et les pièges de l'emploi de la couleur pour les interfaces homme-machine (et la visualisation graphique des données en général).

Si la couleur a très tôt été utilisée dans l'animation d'algorithme, comme dans BALSA, elle n'a été employée que sous la forme d'un attribut statique. Brown et Herchberger [BRO 91b] déterminent cinq façons d'employer efficacement la couleur :

- Pour encoder l'état des structures de données, c'est-à-dire utiliser la couleur comme un moyen de transmettre une information, par exemple une variable peut être visualisée avec une couleur plus ou moins claire suivant sa valeur, à condition que cette information ne soit pas fondamentale.

- Pour souligner les domaines intéressants, en changeant par exemple la couleur d'un élément d'une collection qui va ou qui vient d'être affecté.

- Pour unir de multiples représentations, en utilisant toujours la même couleur pour coder une même information présente dans plusieurs vues.

- Pour accentuer la visualisation du comportement, en choisissant, par exemple, lors d'un parcours d'arbre une couleur pour la descente et une couleur pour la montée.

- Pour capturer l'historique de l'exécution, en employant un spectre de couleur, c'est à dire associer au cours de l'exécution d'un programme une couleur différente aux informations représentatives de l'histoire du programme suivant l'ordre de leur apparition.

II.5.3.4 - Le son

Un logiciel de visualisation a pour objet de former des images mentales du comportement; des structures et des fonctionnalités des programmes en observant des représentations d'un programme ou d'un algorithme. Ces représentations sont traditionnellement, textuelles ou graphiques, pour les systèmes de visualisation de programme. Cependant, des informations acoustiques constituent une forme de représentation d'un programme, car elles peuvent aussi former des images dans l'esprit du programmeur ; elles doivent donc être considérées comme faisant partie d'un logiciel de visualisation. De plus, si les capacités des sorties audio, dans les années 80, étaient limitées à un simple beep, les ordinateurs modernes sont maintenant capables de restituer du son digitalisé.

Gaver [GAV 91, MOU 90] et ses collègues ont démontré comment le son pouvait être utilisé pour communiquer des informations complexes. Les travaux sur la sonorisation (appelée plus communément auralisation) couvrent les algorithmes [BRO 91b], les programmes [DIG 92], les interfaces [EDW 88] et les données [SMI 91]. Mais seulement LogoMedia et Zeus ont employé de manière exhaustive et efficace le son dans l'animation d'algorithmes et des données. Brown et Hershberger [BRO 91b] ont déterminé quatre emplois distincts du son :

- Pour renforcer les informations visuelles, ce qui génère des informations qui peuvent être considérées redondantes mais qui peuvent être perçues et exploitées différemment par le visualisateur.

- Pour transmettre le comportement ; par exemple si nous prenons un tri et que nous associons une note à chaque échange et qu'une mesure représente une passe nous percevrons s'il y a plus ou moins d'échanges entre chacune des passes.

- Pour remplacer des informations visuelles (si bien que l'attention visuelle peut être concentrée ailleurs).

- Signaler des conditions exceptionnelles.

Par ailleurs Francioni, Jakson et Albright [FRA 92] ont étudié l'emploi du son dans les programmes parallèles. L'auralisation est alors utilisée plus spécifiquement pour rendre compte des échanges entre les processus, que se soit en nombre en écoutant leur fréquence d'apparition, ou sur leur nature en leur associant une note particulière. De même, une ou plusieurs notes peuvent être utilisées pour avertir du changement d'état des processus.

II.5.3.5 - La dimension

Traditionnellement les systèmes de visualisation ont utilisé la représentation graphique à deux dimensions, bien que quelques travaux récents ont utilisé des projections d'images à trois dimensions sur un écran à deux dimensions. Par exemple, Pavane fournit des outils pour faire des rotations d'images dans l'espace 3-D. Plusieurs techniques existent pour obtenir une vue en trois dimensions, telles la projection, les images polarisées, les images stéréographiques, etc. Certaines visualisations dans ANIM utilisent la polarisation pour fournir une illusion de profondeur binoculaire. L'émergence des systèmes à réalité virtuelle promettent de procurer de meilleures visualisations 3-D et des techniques telles que la transversalité graphique 3-D de Stasko [STA 93] peuvent devenir ordinaires. L'utilisation d'une visualisation 3-D pour montrer une information qui est naturellement à trois dimensions n'est pas nécessairement la plus efficace ; le système Zeus-3D de Brown et Najork [BRO 93], l'animation de tri de Stasko dans POLKA-3D [STA 93], et le récent travail avec Pavane [ROM 92a] (Cf. figure 2.13) sont des exemples de l'emploi de la dimension supplémentaire pour coder une information non dimensionnelle. Les visualisations 3-D dans ANIM se résument simplement à une projection en trois dimensions de données naturellement 3-D.

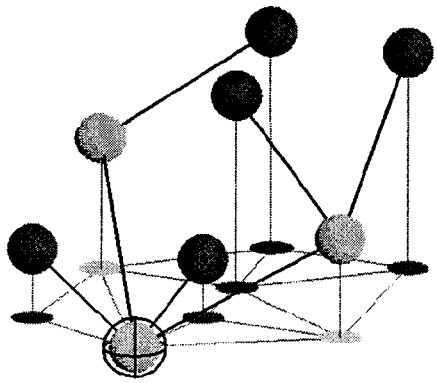


***Figure 2.13** : Animation dans Pavane d'un algorithme de parcourt de graphe en 3-D dont le calcul des distances pour un noeud particulier est visualisé par un cercle* [1].

Si ce style de visualisation possède certains avantages, son emploi ne se justifie que pour la représentation d'informations de quelques algorithmes particuliers. En fait, la plupart des logiciels emploie ce style de représentation d'un point de vue esthétique ce qui ne justifie pas, a priori, le coût de développement d'un tel modèle graphique. Par exemple, si on considère l'algorithme de parcourt de graphe de la figure 2.13, l'ombre du graphe visualisé en 2-D peut suffir à représenter le comportement de l'algorithme (Cf. chapitre V.3).

II.5.4 - Gestion des éléments graphiques

Après avoir déterminé les éléments graphiques d'une visualisation, nous allons nous intéresser plus particulièrement aux actions que nous pourrons appliquer à ceux-ci pour nous permettre d'effectuer une animation, c'est-à-dire passer d'une représentation statique à une représentation dynamique.

II.5.4.1 - L'animation

Nous définissons un événement visuel comme la création, la destruction, ou la modification d'un objet graphique. Le résultat de cette transformation peut être réalisé instantanément, mais la plupart des systèmes fournissent des mécanismes pour animer ce changement. Par exemple, si la taille d'un objet change de deux unités à huit unités, le système

[1] Les noeuds du graphe sont représentés sous forme de sphères ; les sentiers et les distances entre les noeuds sont visualisés par des lignes. Les couleurs représentent l'état d'un noeud lors de l'exécution de l'algorithme, passé par le noeud (gris clair), pas passé par le noeud (gris foncé). L'algorithme animé par Pavane consiste, en fait, à trouver le plus court chemin en passant une seule fois par tous les noeuds du graphe.

renvoie cet événement comme une séquence d'images avec une taille variant de deux, trois, quatre, jusqu'à huit unités. Si les images sont exposées suffisamment rapidement, le résultat apparaîtra comme une douce croissance de l'objet. En fait l'animation repose sur une singularité des capacités du système visuel qui consiste à envoyer à notre système perceptif des stimulus successifs suffisamment rapprochés (au plus 0.1 seconde entre 2 stimulus) pour qu'ils soient perçus comme un stimulus unique [THA 85].

II.5.4.1.1 - L'animation traditionnelle

L'animation traditionnelle consiste à créer des images intermédiaires qui permettront de simuler le mouvement entres les dessins clés. Il suffira d'enregistrer toutes ces images une par une et de les projeter suffisamment rapidement de manière successive pour donner l'effet d'une animation continue. Au cinéma ce type d'animation a été largement employé en projetant les unes après les autres 25 images fixes par seconde par l'intermédiaire d'une caméra.

II.5.4.1.2 - L'animation par interpolation

Le principe d'une telle animation est de définir les deux images clés d'une séquence et de laisser à un ordinateur la charge de calculer l'ensemble des images intermédiaires.

Les techniques de calcul utilisées sont des interpolations linéaires ou à base de splines (Cf. figure 2.14).

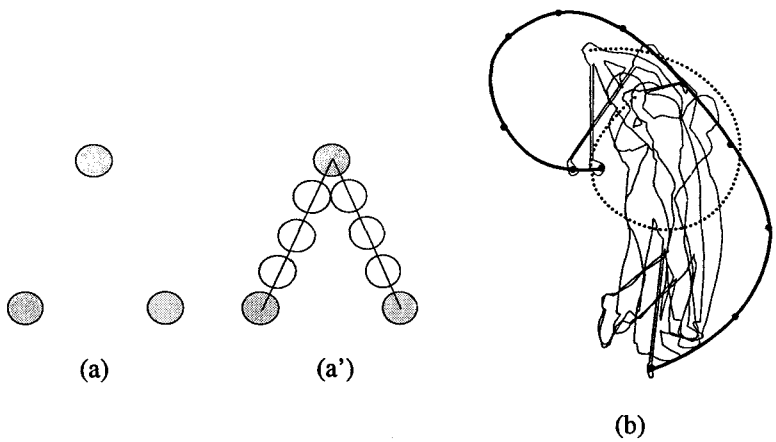


***Figure 2.14** : Interpolation de mouvement : (a) définition de trois images-clés positionnant une balle, (a') obtention des images intermédiaire par interpolation linéaire, (b) définitions de trajectoires à base de splines*

Dans l'interpolation linéaire, les déplacements générés sont essentiellement rectilignes. Les premiers travaux sur l'interpolation linéaire ont été menés par Burtnyk

[BUR 71] et par Martinez [MAR 77] pour l'interpolation linéaire par morceaux. Ces méthodes possèdent l'inconvénient d'introduire des discontinuités d'un mouvement à l'autre.

Un autre procédé, plus général et plus simple, consiste à définir des trajectoires joignant deux dessins clés successifs. Cette méthode est particulièrement intéressante si elle est liée à un dispositif de saisie automatique des données. La vitesse de déplacement le long d'une trajectoire peut alors être contrôlée par des fonctions du type : $s = \sum_{t=0}^{t=+\infty} s(t)$, définissant les positions successives des points sur la courbe (s est l'abscisse curviligne sur la trajectoire et t le temps ou le numéro de l'image).

Baecker [BAE 69] a été le premier à proposer une solution de ce type, en définissant une double description du mouvement : l'une est une trajectoire spatiale donnée à l'aide d'une courbe géométrique, l'autre décrit la cinématique du mouvement grâce aux P-courbes (Cf. figure 2.15). Cette technique a été employée dans le système Genesys. La méthode des P-courbes a été généralisée par Reeves [REE 81].

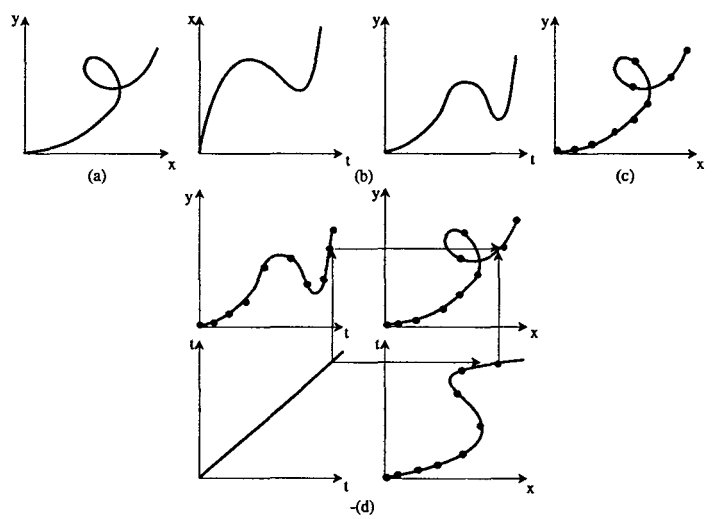


***Figure 2.15** : Construction d'une P-courbe, (a) définition d'un chemin, (b) composants x et y fonction du temps, (c) courbe originale marqué d'indicateur égale au pas de temps, (d) construction de la P-courbe à partir des différentes fonctions.*

L'interpolation à base de splines est une méthode qui a été proposée par Shoemake [SHO 85]. Il s'agit d'utiliser des B-splines du quatrième ordre qui permettent d'assurer la continuité des courbes, mais aussi des tangentes et des courbures qui correspondent aux notions de vitesse et d'accélération et de décélération des objets.

Bret dans sa thèse [BRE 84] propose une méthode permettant à la courbe recherchée de passer par tous les points de contrôle fournis par l'animateur.

II.5.4.1.3 - L'animation par langage de programmation

Un animateur ne trouvant pas les outils nécessaires pour obtenir un effet spécial ou réaliser un mouvement particulier doit programmer l'animation voulue dans un langage de programmation classique du type C, Pascal, Lisp, etc. Cette méthode demande de bonnes connaissances en programmation ; elle est lourde à mettre en oeuvre car l'animateur ne voit le résultat de son animation que lorsque le programme est complet et elle est coûteuse en temps de développement.

II.5.4.1.4 - L'animation à base de scripts ou de langage graphique

Des systèmes basés sur l'écriture de scripts ou sur l'implémentation de langage graphique ont été développés dans le but de fournir une méthode d'animation plus flexible.

Les éditeurs de scripts se présentent sous forme de langages de programmation spécialisés de haut niveau incluant toutes les possibilités des langages (notion de procédure, de récursivité, de structure de contrôle, de variables locales) pour modéliser l'animation d'une scène. Ces éditeurs décrivent à la fois les objets géométriques, les conditions de visualisation et l'animation de séquence. Le script d'une séquence animée peut être converti en une suite d'images par un interpréteur. Il est aussi possible de réaliser des bibliothèques d'objets et de mouvements pouvant être réutilisées à tout moment.

La première forme de script consiste à utiliser une simple notation sous forme de liste linéaire présentée par [CAT 72] dont chaque événement dans l'animation est déterminé par deux nombres correspondant au début et à la fin de celui-ci et une action sur un objet déterminé. Par exemple, "42, 53, Rotate 'Palm', 1, 30" effectue une rotation de l'objet 'Palm' de 1 à 30 degré dans le cadre de l'événement 42, 53. Scefo (SCEne Format) [STR 88], par exemple, a utilisé quelques aspects de la notation de liste linéaire, mais inclut également une notation de groupes, d'objets hiérarchiques, de supports d'abstraction des changements (appelée actions) et plusieurs constructions de programmation de haut niveau (variables, flux de contrôle, et évaluation d'expression) qui le distingue d'une simple liste linéaire. Un script Scefo décrit seulement une animation ; les objets individuels dans le script peuvent être interprétés, avec plusieurs autres, de manière indépendante.

Le langage d'animation DIAL (DIagrammatic Animation Language) [FEI 82] garde également plusieurs caractéristiques de la notation linéaire mais la visualisation des séquences d'événements dans une animation est traitée de manière parallèle. Une barre verticale indique l'introduction d'une action, et des traits indiquent le temps que doit durer cette action. Deux types de ligne définissent les actions : une ligne de définition d'événement (utilisant une notation linéaire) associée a un nom avec un "retour" à l'action d'animation, et une ligne définissant sa durée (Cf. figure 2.16). Le système S-Dynamics [SYM 85] combine le langage d'animation DIAL et les P-courbes.

```
      Lecture d'un objet dans un fichier et assignation du nom "block"
! getsurf "block.d" 5 5 "block"
      Definition d'une fenêtre dans le plan xy
! window -20 20 -20 20
      Definition de deux actions, (1) une translation,
% t1 translate "block" 10 0 0
      (2) une rotation de 360 degrée dans le plan xy
% r1 rotate "block" 0 1 360

      Description du temps de chaque action
t1 |--------          |----------
r1          |------------
```

***Figure 2.16** : Définition d'un script DIAL typique (les lignes commençant par un blanc sont des commentaires)*

Mais le plus connu des systèmes à base de script est ASAS (Actor Script Animation System) [REY 82] qui utilise le concept d'acteur introduit par Hewitt [HEW 77] et a été implémenté en Lisp. Si le script est un programme principal dont le déroulement produit l'animation complète d'une scène, un acteur peut être considéré comme un sous-programme généré par le script qui contient un certain nombre de paramètres qui peuvent entretenir des relations avec d'autres acteurs et dont le but est de contrôler l'animation d'un objet particulier. Cet acteur est exécuté une fois que chaque image est produite dans la boucle "Animate". Dans Asas le script peut contenir des primitives (géométriques, de couleur, d'éclairage, de prise de vue, etc.) et des opérations de modification (déplacer, tourner, changement d'échelle, etc.).

II.5.4.1.5 - L'animation dans la visualisation d'algorithmes

Tango a des facilités sophistiquées pour définir des animations dites "lisses", en utilisant un mécanisme qui applique un nouveau paradigme "chemin transition" à un objet. Ce modèle introduit quatre types de données abstraites pour définir des transitions qui sont : une définition d'un mouvement dans un certain espace (utilisant une algèbre de chemins), le changement de coordonnées, de couleur et de visibilité (Cf. figure 2.17). Pavane fournit également des mécanismes pour générer des événements visuels en attribuant des valeurs de temps variables aux attributs des objets.

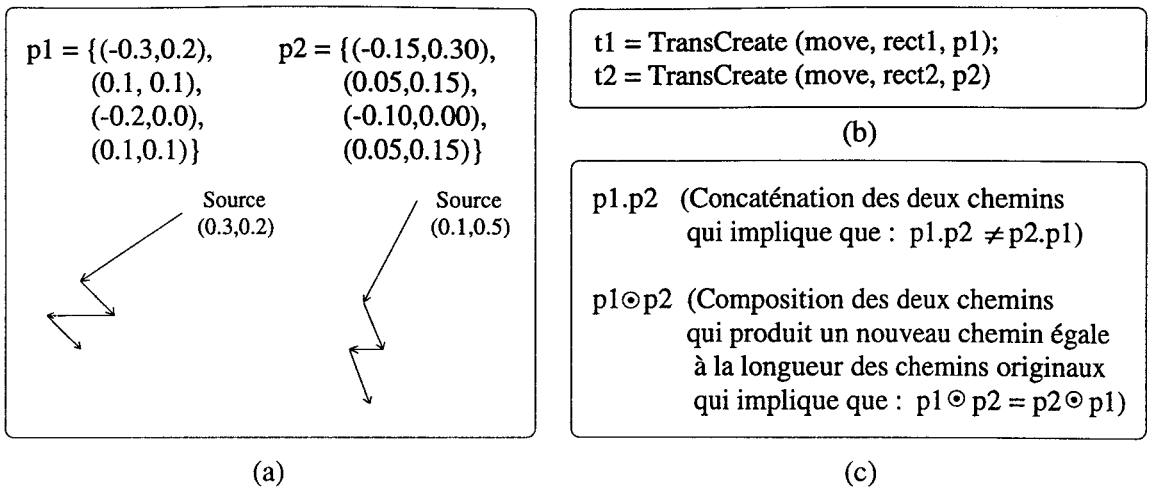


***Figure 2.17** : Composants d'une animation du système Tango : (a) définition de deux chemins "p1, p2", (b) création de deux transitions "t1, t2" à partir des chemins "p1, p2", (c) exemple d'algèbre défini sur les chemins*

II.5.4.2 - Le positionnement des objets dans une vue

Les systèmes de visualisation de programmes combinent les objets graphiques en établissant des relations géométriques entre leurs positions dans la vue, qui est un espace à deux ou trois dimensions. En générant l'image, l'animateur sélectionne un point de vue dont il visualise le contenu.

Les systèmes utilisent une ou deux approches principales pour le positionnement des objets. La première place les objets selon les coordonnées absolues, avec chaque objet indépendant de tous les autres. L'autre approche, généralement appelée positionnement à base de contrainte, spécifie les positions des objets qui constituent une image comme des relations qui contraignent un objet à se positionner par rapport à un, ou plusieurs autres objets (Cf. figure 2.18). Ainsi, cette méthode est souvent utilisée pour construire les objets composés ; la position des composants est en relation avec celle de l'objet composé.

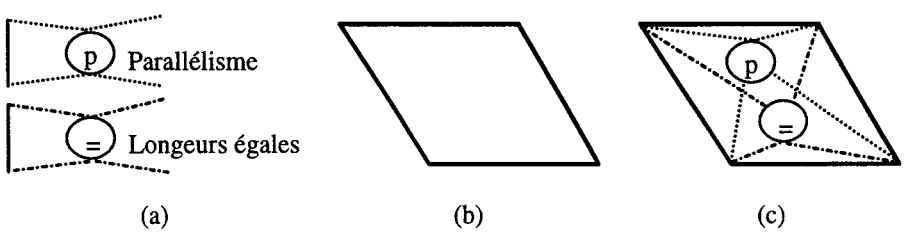


***Figure 2.18** : Définition d'un contrainte : (a) les opérateurs, (b) l'image contrainte ,(c) satisfaction des contraintes*

Balsa, Pavane et Tango, pour ne citer qu'eux, permettent des formes limitées de positionnement à base de contraintes, mais le système Animus de Duisberg [DUI 86a, 86b] a probablement les capacités les plus sophistiquées en ce domaine. Ce système est bâti sous Smalltalk, comme une extension au langage à contrainte appelé Thinglab [BOR 79] dont les contraintes sont utilisées pour maintenir une cohérence entre la structure d'un programme et les vues de l'interface qui sont présentées. La différence entre Thinglab et Animus est que ce dernier utilise des contraintes temporelles à évolution discrète (encore appelées déclencheurs) pour modéliser un comportement dynamique et donc effectuer une animation graphique. Ces contraintes s'appliquent à une variable particulière, le temps, qui est global à toute une animation et les déclencheurs décrivent la réponse d'un objet particulier à un événement. Le propriétaire de la contrainte stocke d'abord les événements dans un tampon, puis effectue les actions associées. Quant au mécanisme de satisfaction de ces contraintes, il utilise une planification avec compilation avant leurs exécutions. En fait, les contraintes n'affectent pas directement les valeurs, mais les événements sont mis en attente dans une file. Tous les protocoles de la gestion du temps sont regroupés dans une seule classe, Anima, qui est le coeur de la simulation, et qui gère la file des événements ainsi que la progression du temps dans la simulation. Chaque classe de la simulation est doublée d'une autre classe qui compile ses contraintes temporelles, et qui répond essentiellement aux messages de progression temporelle de la simulation par pas de temps en activant les procédures attachées aux événements ayant une date située dans le pas de temps. Tous les événements étant datés, c'est donc l'ordre de datation qui définit les propriétés de déclenchements des procédures associées. Ce principe ne nous permet que des formes d'animation et d'interactions limitées car nous n'affectons pas directement les valeurs. De plus, les systèmes à base de contraintes sont coûteux en temps de calcul, parce que la détermination des valeurs des attributs peut entraîner la résolution de grands systèmes d'équations. Cependant, nous nous attendons à ce que ce type de système prenne une place plus prépondérante dans un futur proche avec l'apparition de bibliothèques spécialisées, telle que celle de la société ILOG sous C++.

II.5.5 - L'interface visuelle

L'emploi de visualisations multiples, ayant leur propre collection d'objets graphiques, peut se faire dans des fenêtres ou des vues séparées sur l'écran (Cf. figure 2.19).

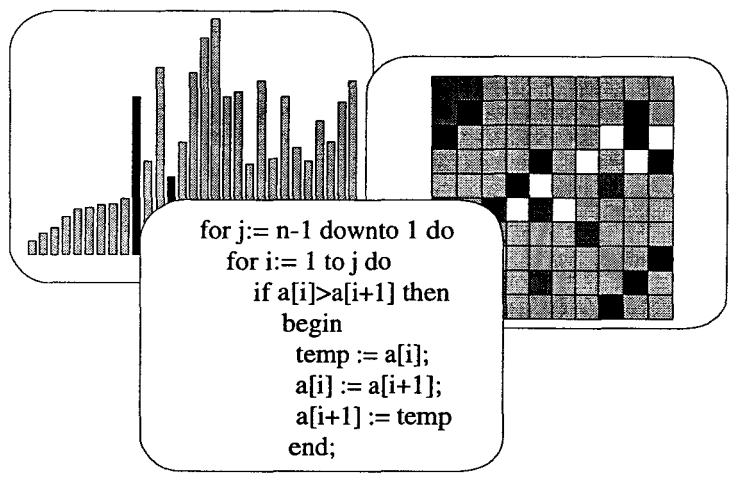


Figure 2.19 : Visualisations multiples d'informations

Ces visualisations multiples sont souvent employées pour montrer différentes représentations visuelles d'un calcul. La comparaison de ces différentes visualisations peut conduire à une plus grande compréhension du comportement du programme. Les visualisations multiples peuvent également limiter le nombre d'informations montrées dans chaque représentation.

II.5.5.1 - Les fenêtres multiples

Smalltalk [GOL 84] introduit non seulement un nouveau langage prolongeant l'approche orientée objet de Simula-67, mais également une nouvelle interface hautement visuelle pour l'utilisateur. Alan Kay, l'initiateur de la recherche pour l'environnement de programmation de Smalltalk-76, a conçu un nouveau paradigme pour l'interface utilisateur qu'il appelle "chevauchement de fenêtres". Les aspects fondamentaux du paradigme de Kay sont :

- des visualisations associées à plusieurs tâches de l'utilisateur peuvent être vues simultanément,
- le changement entre ces tâches peut être déclenché par l'appui d'un bouton,
- aucune information ne peut être perdue dans l'exécution d'un changement,
- et l'espace de l'écran peut être utilisé économiquement.

Ce paradigme peut servir de base pour ce que Kay appelle un "environnement intégré", dans lequel la distinction entre le système d'exploitation et une application peut disparaître jusqu'à ce que chaque morceau de logiciel soit capable d'appliquer un morceau d'information. Son modèle contient les fondements pour définir les paradigmes d'une interface utilisateur actuelle, non seulement pour des environnements de langage, mais encore pour définir les structures des systèmes, dont celui de l'environnement familier "Mach" implanté sur les ordinateurs Macintosh d'Apple.

Une alternative pour définir le paradigme sur le fenêtrage a été introduite dans Cedar [SWI 86], un langage de la famille Algol synthétisant plusieurs environnements, qui différencie son modèle sur quelques détails et sur la philosophie pour gérer les fenêtres. Celles-ci sont appelées "visionneuse" et sont par défaut mises en mosaïque plutôt que de les chevaucher partiellement ou totalement. L'avantage de cette méthode est d'automatiser le placement des "visionneuses" mais a pour inconvénient de ne pas pouvoir maximiser la visibilité de ces contenus.

II.5.5.2 - Les vues multiples

Si Smalltalk a introduit le concept de fenêtres multiples et permet par l'intermédiaire du modèle MVC et du mécanisme de dépendance de créer des vues multiples (Cf. chapitre III.3.1), Pecan [REI 85] utilise et introduit le concept de manière plus généralisée. La distinction de ce modèle est que les vues multiples partagent une représentation commune des aspects internes de la même donnée. Quand des aspects de cette donnée changent, alors toutes les vues changent simultanément pour refléter ce changement. L'idée est qu'en représentant simultanément des données de plusieurs manières, un utilisateur peut choisir les vues qui lui semblent les plus utiles ou les plus représentatives à un instant particulier.

II.6 - LES INTERACTIONS

La prise d'informations par des moyens traditionnels, tels que les livres, offrent à l'utilisateur des formes limitées d'interaction. Ainsi, dans un livre le lecteur peut sélectionner quelques pages à examiner mais ne peut changer les contenus ou le formatage d'une page. Par contre, les ordinateurs permettent de présenter et de visualiser énormément d'informations et d'utiliser un large ensemble d'interaction. D'autre part, en plus de sélectionner une partie de l'information à examiner, le visualisateur peut souvent modifier l'information ou la façon dont elle est présentée avec des techniques différentes. Nous noterons, également, que le niveau de convivialité d'un logiciel est déterminé par le style d'interactivité utilisé [COU 90a]. La hiérarchie de la catégorie interaction est présentée à la figure 2.20.

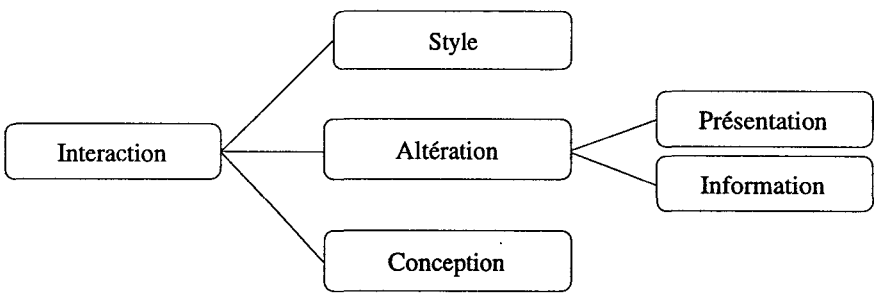


***Figure 2.20** : Sous catégories de la catégorie interaction.*

II.6.1 - Style d'interaction

Cette catégorie regroupe les moyens par lesquels nous allons pouvoir transmettre une commande et de ce fait contrôler le système. Nous trouvons donc les boutons présentés à l'écran, les menus déroulants, les traitements d'une ligne de commande (textuelle) ou des macro commandes qui exécutent une séquence de traitements préétablis par l'utilisateur. Un système peut utiliser une ou plusieurs de ces techniques dont le choix est dépendant de l'action requise.

II.6.2 - Interaction d'altération

Nous avons distingué, dans cette catégorie, les modifications qui agissent sur la forme des représentations des aspects du programme ainsi que son mode d'exécution et celles qui affectent les différentes informations de l'algorithme.

II.6.2.1 - Interactions sur la forme de la présentation

Les interactions typiques concernent, d'une part, les fenêtres ou les vues en permettant à l'utilisateur de choisir l'information visualisée, de la positionner où il la désire sur l'écran et avec une taille d'affichage variable. Nous pouvons inclure tous les modes de fonctionnement de la simulation qui conditionnent l'information affichée sans l'altérer. Par exemple, le contrôle de la vitesse avec laquelle les images sont exposées, les sauts d'événement avec définition d'un point d'arrêt, etc. La plupart des systèmes de visualisation d'algorithmes permettent au moins à l'utilisateur d'arrêter et de commencer la visualisation du programme, tandis que BALSA, Zeus, Pascal Génie, et UWPI ont tous un contrôle explicite de la vitesse. BALSA et Zeus peuvent également changer le sens de la visualisation, c'est-à-dire exécuter le programme en arrière, ce qui peut permettre de visualiser de nouveau une information mal comprise. TPM avec sa possibilité de "remonter au début" permet également une forme de contrôle sur la direction d'exécution.

D'autres interactions sont très utiles quand nous considérons de très larges prog0rammes ou de grands ensembles de données [BAL 96]. Eisenstadt [EIS 90] suggère que ce niveau d'interaction peut être atteint par des changements de résolution, d'échelles, de compression, de sélectivité et d'abstraction. En fait, peu de systèmes de visualisation d'algorithmes autres que SEE et TPM supportent un large niveau d'interaction. Par exemple nous pouvons trouver d'autres techniques comme l'élision [PRI 93] qui consiste pour l'utilisateur à pouvoir élider (2) une information ou supprimer un détail de la visualisation et ne présenter que l'information pertinente. Nous pouvons remarquer que cette technique peut, également, s'appliquer à l'auralisation, ainsi l'élision temporelle (monter en vitesse) peut supprimer des détails auditifs. L'élision est surtout employée avec de grands problèmes, pour lesquels l'ensemble des données ne peut être visualisé simultanément et cette technique a été employée par Pascal Génie, SEE et TPM.

(2) L'élision a ici le sens de cacher une partie de l'information initiale en contractant visuellement celle-ci.

II.6.2.2 - Interactions sur les aspects du programme

Deux grands types d'informations peuvent être altérés lors de l'exécution d'un programme, le code et les données du programme. La modification du code source d'un programme oblige la simulation à recompiler celui-ci avant de le réexécuter. Ce mécanisme ne peut être obtenu par tous les systèmes de visualisation d'algorithmes ; il faut en effet que l'animation reflète le nouveau code généré sans être obligé de redéfinir celle-ci. Ainsi, BALSA et Zeus qui gèrent les événements visuels par annotation sont des systèmes qui par leurs principes ne peuvent permettre ce type d'interaction.

En fait, les données du programme peuvent être changées de deux manières différentes ; soit en changeant de manière textuelle la valeur de la variable considérée par l'intermédiaire d'un inspecteur qui pointe sur la case mémoire de celle-ci ; soit en agissant, par exemple, par l'intermédiaire d'une souris sur sa représentation graphique. Cette dernière technique consiste à sélectionner un des objets graphiques d'une représentation et à le changer de position. Mais alors, ce mouvement peut être interprété de deux manières différentes. Soit, nous voulons améliorer la présentation et dans ce cas cette interaction ne concerne que la représentation qui doit prendre en compte cette nouvelle position pour générer les images suivantes. Soit, l'utilisateur veut modifier la ou les données que l'objet graphique représente. Il faut donc que l'animateur puisse lever cette ambiguïté d'interaction avec l'image.

D'autre part, quelle que soit la technique d'altération employée, il faut que le changement de la valeur d'une donnée soit communiqué à l'algorithme pour qu'il puisse prendre en compte cette nouvelle valeur.

II.6.3 - Interactivité de conception

Le but d'un système interactif n'est pas seulement de permettre le dialogue homme-machine de manière conviviale pendant la phase d'utilisation du logiciel. Mais il doit, également, faciliter le dialogue pendant la phase de conception de la visualisation de l'algorithme.

Certains logiciels comme UWPI (Cf. chapitre II.4.2.4) ont automatisé la création de leurs visualisations. Mais ceux-ci présentent certains défauts dont le plus gênant est que les animations réalisées peuvent être erronées car cette technique repose, obligatoirement, sur une bibliothèque d'animation prédéfinie et est limitée à un certain type de donnée.

Une autre technique consiste à semi-automatiser cette création pour nous permettre de concevoir de nouvelles animations, si le besoin s'en fait sentir, sans être un expert du système. Le principe est de constituer des visualisations sans connaître aucune des spécificités

particulières du logiciel (langage d'animation, nom des objets graphiques, paramètres d'animations, etc.). Pour ce faire, nous devons pouvoir associer des objets graphiques à des méthodes d'animations en adoptant un style interactif et convivial. Il faut donc avoir des outils qui nous permettent de construire une visualisation graphique et une animation de manière transparente, vis-à-vis des mécanismes mis en jeu, et de la lier à l'algorithme de la même façon. Le système Tango présente deux outils appelés DANCE (Demonstrational Animation CrEation) [STA 89] et FIELD qui permettent respectivement d'associer un objet graphique à un "chemin" avec une "transition" et d'introduire des annotations dans le code source. Mais ces outils ont été construits pour permettre de traiter les événements par annotation et héritent des inconvénients qui lui son inhérents.

Les mérites de la manipulation directe ou WYSIWYG ("What You See I What You Get") peuvent être trouvés dans la Programmation par l'exemple [HAL 84], par répétition [GOU 84], et par les systèmes de démonstration [RUB 89], ARK [SMI 86, 87], Juno [NEL 85], Tweedle [ASE 87], Peridot [MYE 87], Graffiti [BEA 88], et les constructeurs d'interface [CAR 88]. En fait, la popularité du WYSIWYG et de toutes les formes de programmation comparable est due au "Feed-back" immédiat donné à l'utilisateur qui peut visualiser immédiatement ses actions, visualiser directement ses erreurs et les corriger, etc.

II.7 - CONCLUSION

La décomposition des modèles de visualisation d'algorithmes nous a permis de déterminer les différentes sous-catégories afférentes à ces composants ainsi que les techniques employées pour les implémenter. Celles-ci sont classifiées et explicitées dans le tableau 2.2.

Aspects visualisés	Code	Peut être présenté sous forme d'arbres, d'organigrammes, de diagrammes ou textuelle.
	Données	Peuvent être présentées sous forme textuelle ou graphique et évoluer de manière statique ou dynamique.
	Etat de contrôle	Présente le processeur d'un programme, la ligne ou l'expression évaluée ou les modules exécutés.
	Comportement	Présente les interactions et l'évolution de plusieurs informations.

***Tableau 2.2 :** Composants des systèmes d'animation d'algorithmes et techniques leur afférente*

Style de représentation	Directe	Effectue une présentation telle qu'il y a une correspondance directe entre l'information et son image ou inversement.
	Structurale	Effectue une présentation en cachant ou soulignant certaines informations.
	Synthèse	Effectue une présentation du ou d'une partie d'un programme sous la forme d'une image éloignée de celle de ces données.
Méthode de gestion des informations	Post-mortem	Recueille les informations sous la forme d'une base de données et effectue ultérieurement ces animations. A pour avantage de rendre l'information, sur un état passé ou futur, disponible à tout moment mais a pour inconvénient de ne pouvoir voir l'effet d'une interaction immédiatement.
	Vivante	Recueille les informations et effectue ces animations au cours de l'exécution du programme. A pour avantage d'autoriser l'utilisateur à interagir avec l'ensemble des informations au cours de l'exécution du programme.
	Annotation	Déclenche une animation en introduisant dans le code de base une instruction supplémentaire. A pour avantage de s'adapter à n'importe quel type d'animation mais a pour inconvénient de requérir davantage de travail et ne permet pas de changer le code d'un programme sans redéfinir son animation
	Déclaration	Déclenche une animation en déclarant différentes relations entre des états du programme et les paramètres d'une image. A pour avantage de découplé le code de son animation ce qui permet de le changer facilement, de définir et d'isoler certains événements indépendamment du code mais à pour inconvénient de ne pouvoir facilement considéré plusieurs événement comme un seul.
	Manipulation	Appelée encore animation par démonstration, elle a pour principe de manipuler les objets graphiques et de capturer les gestes en les attachant à des événements spécifiques du programme. A pour inconvénient de ne pouvoir spécifié (sauf pour des cas spécifiques) l'exact rapport entre le geste et les données ou les événements du programme.

Tableau 2.2 (suite) : Composants des systèmes d'animation d'algorithmes et techniques leur afférente

Forme des informations et de l'interface	Image	L'image bitmap est définie sous la forme d'un tableau de pixels et est gérée point par point. L'image vectorielle est définie sous forme d'objets géométriques et à pour avantage d'être gérer par l'intermédiaire d'un nombre de paramètre limité.
	Vocabulaire graphique	Une image peut être créée à partir d'un seul objet graphique, ou de plusieurs, dont chaque élément est constitué d'un certain nombre de paramètres constituant le vocabulaire graphique (largeur, couleur, etc.). A celui-ci peuvent être ajoutés le son et la dimension de l'espace de visualisation.
	Gestion des éléments	La gestion s'effectue à travers des animations du style cinématographique, interpolation linéaire ou à base de splines entre deux scènes-clés, langage de programmation, script ou de langage graphique. L'inconvénient de ces animations est qu'elle requière des connaissances préalables.
	L'interface visuelle	La représentation des informations peut s'effectuer dans des fenêtres ou des vues multiples.
Interactions	Style	La transmission d'une commande peut se faire par l'intermédiaire de boutons, de menus déroulants, de lignes de commande textuelle ou de macro commandes.
	Altération	Nous pouvons altérer la présentation ou l'information en agissant respectivement sur les vues, la taille d'affichage, les modes de fonctionnement, en élidant une partie de l'information ou en modifiant le code source ou la valeur des données.
	Conception	Introduction d'outilS conviviaux permettant d'automatiser la construction des animations et des représentations.

Tableau 2.2 (suite) : Composants des systèmes d'animation d'algorithmes et techniques leur afférente

Historiquement, les systèmes ont essayé tout d'abord d'améliorer la représentation du code ; puis ils se sont attachés à améliorer celle des données. Ainsi, nous avons eu des systèmes permettant de visualiser graphiquement les données de manière statique puis de manière dynamique avec l'arrivée d'ordinateurs plus performants. Mais les véritables systèmes de visualisation d'algorithmes sont nés avec l'apparition de la visualisation du comportement d'un programme et ont introduit trois manières de représenter les informations. La représentation directe et structurale concerne principalement la visualisation des données et

permet avec la représentation de synthèse de visualiser l'exécution d'un programme par l'intermédiaire d'une image animée représentative du déroulement de son traitement. Ainsi la construction d'une visualisation graphique, abstraite par rapport aux données du programme, montre le comportement et les interactions qu'ont entre elles celles-ci et constitue l'image du problème résolu par l'algorithme. Nous avons également remarqué que suivant les langages de programmation utilisés nous n'allons pas visualiser les mêmes informations car ils utilisent des concepts structurels différents.

Après avoir défini les aspects de la visualisation, nous nous sommes intéressés à la manière d'obtenir les informations et de les transmettre aux différentes représentations pour qu'elles réactualisent leurs affichages. Nous avons, ainsi, des systèmes qui les recueillent sous la forme d'une base de données et ceux qui les analysent en temps réel en affectant leurs représentations ultérieurement ou au cours de l'exécution du programme.

Désirant un système interactif, nous nous sommes intéressés plus particulièrement à cette seconde forme d'obtention et de transmission d'information que nous avons appelée "vivante". Nous avons, dans ce cadre, relevé trois techniques dominantes : par annotation, par déclaration, et par manipulation ou démonstration. La plus intéressante est par déclaration car elle n'entrave pas les capacités d'interactivité du système. La technique par démonstration augure une meilleure convivialité mais elle présente des défauts qui lui sont intrinsèques (Cf. tableau 2.2).

Nous nous sommes ensuite attachés à étudier les différentes formes visuelles et conceptuelles d'une interface. Notre première remarque a été que pour représenter les données ou les comportements d'algorithmes les images symboliques sont plus faciles à gérer que les images numériques et nous a conduit tout naturellement à nous intéresser aux images vectorielles. Celles-ci correspondent, en fait, à des objets graphiques simples ou composés qui ont un vocabulaire graphique associé « dit lexical » en ce qui concerne les paramètres de l'image et « syntaxique » pour permettre de les gérer dynamiquement par le système. Ce vocabulaire permet alors de créer une animation en employant différentes techniques dont les plus intéressantes font appel à un langage graphique évolué. Mais pour créer ces images sans connaître aucune spécificité du système il faut pouvoir associer des objets graphiques et des méthodes d'animation de manière transparente, vis-à-vis des mécanismes mis en jeu.

La gestion de l'espace de travail, quant à elle, dépend de la forme visuelle de l'interface choisie et par conséquent selon que l'on jouit des paradigmes des vues ou du fenêtrage multiple.

Enfin, nous pouvons voir d'après l'évaluation de huit logiciels les plus caractéristiques du tableau 2.3, qu'il n'existe aucun système de visualisation et d'animation d'algorithmes répondant de manière satisfaisante à des contraintes de convivialité ou d'interactivité, de conception ou d'altération. Il est de même, dans ce cadre, très difficile d'obtenir une simulation dont la spécification d'une animation reste simple, générique, et transparente pour l'utilisateur.

	Aspects visualisés				Spécifications					Formes							Interactions		
	Code	Données	Etat de contrôle	Comportement	Vivant	Post Mortem	Annotation	Déclaration	Manipulation	Objets graphiques	Couleur	Son	Dimension	Positionnement	Animation	Fenêtres ou vues multiples	Présentation	Information	Conception
BALSA	◇	◡	◡	◕	✓		✓			◕	△	△	◇	◡	◕	●	◡	△	△
TANGO	◇	◕	◡	◕	✓		✓		✓	◕	◡	△	◇	◡	●	◇	◇	△	◇
ZEUS	◇	◡	◡	◕	✓		✓			◕	◕	◕	◕	◡	◡	●	◡	△	◡
ANIM	△	◡	◇	◡		✓				◡	△	△	◕	◇	◡	◕	◕	◇	◇
Pascal Genie	◡	●	△	◇	✓					◡	△	△	◇	◇	◡	◕	◕	△	△
UWPI	◡	◡	◇	◡	✓					◕	△	△	◇	◇	◡	△	◇	△	△
TPM	◇	◕	△	△	✓	✓				◡	△	△	◇	◇	◡	◡	●	◇	△
PAVANE	△	◡	◕	◕	✓			✓		◕	◕	△	◕	◡	●	●	◡	◇	◇

Symboles	△	◇	◡	◕	●	✓
Signification	Inexsistant ou mauvais	En dessous de Moyen	Moyen	Au dessus de Moyen	Trés bon	Utilisation

***Tableau 2.3** : Evaluation de huit logiciels de visualisation de programme*

Cette incapacité à produire un tel logiciel peut expliquer pourquoi ce type de système est encore aujourd'hui très peu utilisé.

Aussi, tout au long du développement de notre propre système nous nous sommes attachés à satisfaire ces différentes spécificités en développant comme nous allons le voir dans le prochain chapitre des outils et des concepts permettant de :

- visualiser tous les aspects d'un programme,
- obtenir tous les styles de représentation,

- gérer les informations de manière, vivante, indépendante du code et automatique,

- visualiser les informations par l'intermédiaire d'image vectorielle en employant tout le vocabulaire graphique possible,

- animer ces images en associant aux objets graphiques des méthodes d'animation de manière transparente,

- s'adapter à l'utilisateur en gérant l'affichage des informations par l'intermédiaire d'une interface multi-fenêtre, et enfin

- autoriser l'utilisateur à interagir avec le système sur la forme de la présentation, sur les aspects du programme visualisé, etc.

CHAPITRE III

LE SYSTÈME D'ANIMATION D'ALGORITHMES DÉVELOPPÉ

L'animation d'algorithmes est un besoin apparu durant les années 90 et qui a pour objectif de faciliter la compréhension des programmes, d'aider au développement de nouveaux programmes, et d'évaluer des programmes existants à l'aide d'animations graphiques.

Dans ce chapitre, nous introduisons notre logiciel d'animation, en spécifiant une méthodologie de description, de spécification, d'analyse, et de formalisation des éléments employés. Nous allons présenter, pour ce faire, les différents composants que nous avons développés et les techniques mises en oeuvre en décrivant plus particulièrement les trois constituants basiques d'un tel système : l'algorithme, l'animation, et la spécification qui permet de transposer l'exécution de l'algorithme en une animation (Cf. figure 3.1).

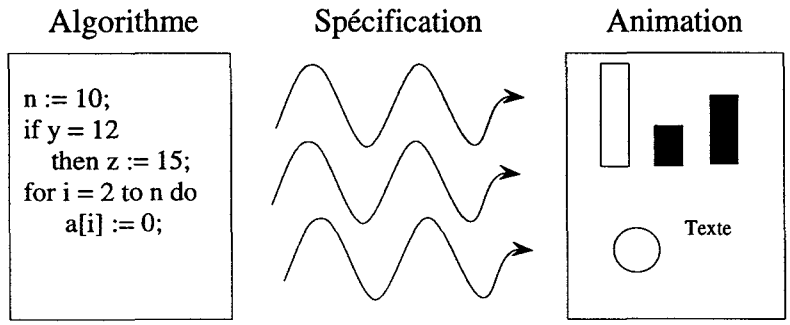


***Figure 3.1** : Les trois constituants basiques employés dans l'animation d'algorithmes.*

III.1 - INTRODUCTION

La détermination des principaux éléments constituant une animation d'algorithmes (cf. chapitre II.2) et l'étude des différentes techniques employées pour chacun de ces constituants m'ont aidé à déterminer un ensemble de fonctionnalités que doit comporter notre système.

Nous avons ainsi défini que les aspects visualisés doivent comprendre le code, les données, l'état de contrôle, et le comportement pour être considérés comme un logiciel permettant de visualiser tous les types d'informations représentatives d'un algorithme.

En fait, si nous voulons vraiment aider l'utilisateur à comprendre le fonctionnement interne des programmes, à en développer de nouveaux ou à en évaluer des existants, nous avons considéré que la visualisation du code de l'algorithme est indissociable de celle de l'animation de ces données ou de son comportement. Par conséquent, nous avons souhaité que les différentes visualisations d'informations liées à l'algorithme ne puissent être obtenues qu'à partir de la présentation de l'exécution du code source.

En outre, ce code peut être présenté sous différentes formes graphiques (arbres, diagrammes, etc.) qui limitent généralement la taille des programmes implémentés. Nous avons donc choisi de nous restreindre à présenter nos algorithmes sous forme textuelle. Cependant pour permettre à l'utilisateur de dissocier l'évaluation de plusieurs instructions contenues dans une seule ligne et ainsi ne pas confondre plusieurs événements de la simulation en un seul, la distinction de la visualisation et du traitement de chacune des expressions ou sous-expressions significatives du programme est nécessaire.

Comme nous l'avons indiqué dans l'introduction générale, un des objectifs de notre système est d'obtenir une simulation interactive. Ceci implique de donner à l'utilisateur la possibilité de modifier les différents paramètres et informations du système au cours de son traitement. Cette interaction est, dans l'animation d'algorithmes, particulièrement contraignante et difficile à mettre en oeuvre pour les données ou le code source du programme, car elle influe directement sur la méthode de spécification employée.

Seule l'approche déclarative, introduite dans le chapitre précédent, qui consiste à unir les valeurs des variables aux attributs des objets graphiques dans l'image finale permet de résoudre cette difficulté. En fait, pour être effective cette association doit s'effectuer indépendamment de l'état des données ou du code source du programme dans la limite où celui-ci garde une certaine cohérence (nom de variable identique, comportement global de la donnée comparable, etc.). L'avantage de cette technique consiste en fait à lancer automatiquement l'animation quand la simulation détecte le changement d'état d'une variable.

L'obtention des informations se fait alors de manière vivante, c'est-à-dire au cours de l'exécution du programme, ce qui permet de prendre en considération leur nouvel état lors d'un changement effectué par l'utilisateur. Mais comme nous allons le voir un mécanisme simple générique doit être élaboré et prendre en compte le style de l'interface visuelle adoptée.

Dans un environnement convivial tel que nous voulons l'élaborer, l'utilisateur doit pouvoir visualiser au cours de la simulation n'importe quelle information. Aussi, la représentation graphique des données doit être obtenue instantanément et indépendamment de l'algorithme considéré. Par conséquent nous allons voir que, pour que les informations connaissent automatiquement leur image, l'objet graphique devra être rattaché soit à la donnée qu'il est censé représenter soit à l'algorithme pour une représentation de synthèse. Cependant si une information n'a pas d'animation graphique associée, son affichage devra s'effectuer sous forme textuelle.

En ce qui concerne la visualisation graphique des différentes informations, nous devons pouvoir les élaborer en adoptant toutes les formes possibles d'abstractions (directes, structurales ou de synthèse, Cf. chapitre II.3.2).

Les représentations seront en fait formées d'images symboliques décrites sous forme vectorielle. Celles-ci peuvent être également constituées d'une composition d'images simples dont la construction doit se faire par l'emploi d'outils conviviaux et d'un modèle graphique générique nous permettant d'associer aux objets graphiques des méthodes d'animations de manière transparente. Pour simplifier leur élaboration et obtenir un modèle de représentation fiable, en éliminant certaines sources d'erreurs, nous avons choisi de nous restreindre à un espace à deux dimensions ce qui n'est pas pénalisant et est suffisant pour représenter la plupart des données et les comportements des algorithmes.

L'affichage des informations d'un algorithme s'effectuera donc sous forme d'animations graphiques, ou par défaut, sous forme textuelle.

D'autre part, pour permettre aux utilisateurs novices en programmation objet ou ne connaissant pas la syntaxe du langage Smalltalk d'écrire et d'animer leur propre programme nous devons leur permettre d'implémenter leurs algorithmes dans un langage structuré de type Pascal (cf. Annexe A).

Aussi, chacun des éléments développés pour la simulation et la visualisation des informations de l'algorithme doit prendre en compte les contraintes liées au langage utilisé (Smalltalk ou Pascal) ou s'en détacher le plus possible.

En résumé, chacun des choix effectués et des techniques employées tout au long de cette thèse ont été principalement motivés par le désir d'obtenir une simulation conviviale, interactive et générique ; que ce soit pour l'écriture des algorithmes ou pour la création des animations graphiques [VAN 95].

Mais avant d'exposer les solutions et les outils développés, nous allons préalablement présenter les caractéristiques de l'interface visuelle de notre système qui influence directement la gestion et le style d'affichage des informations.

III.2 - L'INTERFACE DE VISUALISATION

Comme nous l'avons spécifié en introduction, l'animation d'algorithme s'adresse à un ensemble d'utilisateurs différents par leur culture ou leur connaissance personnelle.

Ainsi, chaque personne, devant un problème particulier, ne raisonne pas et ne réagit pas de la même manière ; elle peut requérir un besoin d'informations spécifique. Aussi, la visualisation de la valeur d'une variable ou d'une expression peut paraître obsolète pour un utilisateur et problématique pour un autre ; il faut donc laisser à l'utilisateur le choix d'organiser son propre scénario.

Pour ce faire, chaque type de représentation dans l'animation d'algorithmes doit pouvoir être obtenu de manière indépendante et autonome.

Ainsi, la visualisation des informations du système a été basée sur l'environnement multi-fenêtre de Smalltalk qui nous permet de présenter le programme, les données, etc., dans des fenêtres autonomes. Celles-ci possèdent l'ensemble des commandes basiques du système de fenêtrage (fermeture, mise en icône, modification de sa taille, etc.) sélectionnable par un menu fugitif. D'autre part, l'interface visuelle de Smalltalk permet de superposer des fenêtres en contrôlant plusieurs plans d'affichage.

Le système peut alors s'adapter à son interlocuteur et visualiser s'il le désire les différentes informations qu'il estime utiles pour son travail et sa compréhension. Ceci est réalisé en gérant l'affichage de manière conviviale et fonctionnelle (c'est-à-dire adapté aux fonctions demandées) tout en permettant à l'utilisateur de concevoir interactivement sa propre visualisation ou sa propre perception de l'exécution par l'intermédiaire des différentes fonctionnalités liées au système de fenêtrage.

III.2.1 - Les aspects de visualisation d'un algorithme

Chaque information visuelle devant être obtenue dans des fenêtres autonomes, nous avons déterminé les diverses représentations que doit fournir notre système d'animation d'algorithmes. Ainsi, sachant qu'un programme est caractérisé par son code source, par les données manipulées, par les expressions exécutées et enfin par son comportement, nous pouvons associer une ou plusieurs représentations pour chacune de ces caractéristiques.

Concrètement, celles-ci ont été regroupées en trois principaux aspects de visualisation.

- Présentation du programme sous sa forme textuelle nous permettant de visualiser l'expression en cours d'évaluation lors de l'exécution de l'algorithme (Cf. figure 3.2).

```
procedure exchange;
var tempo : real;

begin
    tempo := aStream[h];
    aStream[h] := aStream[l];
    aStream[l] := tempo
end;
```

Figure 3.2 : Présentation d'un programme.

- Présentation des données du programme nous permettant de visualiser l'état des variables à un instant donné. Celles-ci peuvent être visualisées sous une forme textuelle ou graphique dans des fenêtres autonomes qui conservent une relation simple avec la fenêtre de simulation pour permettre une remise à jour de leurs représentations respectives (Cf. figure 3.3).

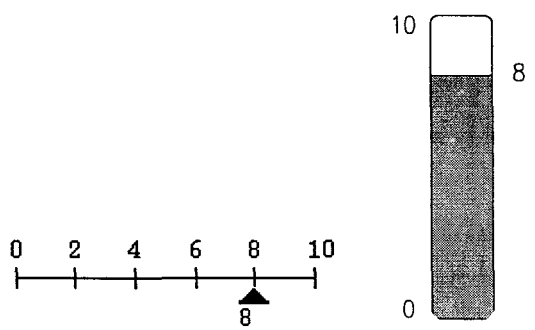


Figure 3.3 : Deux exemples de représentation pour une donnée entière ou réelle.

- Présentation du comportement global ou de synthèse du programme. Celle-ci nous permet de faire abstraction des structures de données employées pour ne représenter que la métaphore du monde réel, c'est-à-dire l'image que l'on peut se faire du problème traité (Cf. figure 3.4). Celle-ci se caractérise, généralement comme nous le verrons ultérieurement, par l'adjonction de plusieurs structures de données pour une représentation globale du fonctionnement de l'algorithme. Aussi, l'animation peut être dans ce cadre utile pour comprendre les différentes interactions qui peuvent exister entre plusieurs structures de données.

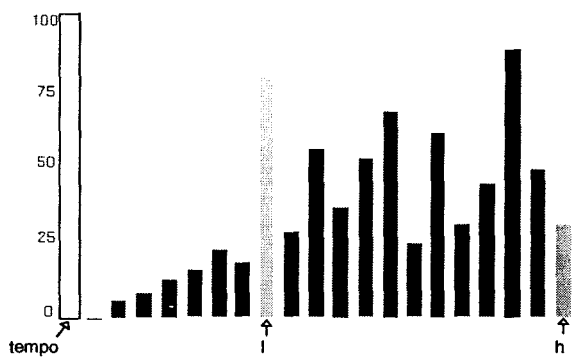


***Figure 3.4** : Présentation de synthèse d'une liste triée avec deux éléments pointés (l & h) et une variable temporaire (tempo) pour visualiser l'échange entre les deux données.*

D'autres informations peuvent être présentées pour non pas comprendre la fonctionnalité des programmes mais pour les évaluer en performance et les comparer ou comprendre comment une donnée est maniée par l'ordinateur.

Toutes ces informations étant gérées et présentées dans l'environnement multi-fenêtré de Smalltalk font naturellement appel aux concepts et outils développés autour de ce langage pour répondre aux objectifs que nous nous sommes fixés (Cf. chapitre II.7).

Ainsi la gestion des événements du système, mettant à jour les représentations au cours de la simulation, est bâtie sur une extension du concept de dépendance et du modèle d'architecture d'interface M.V.C. (Modèle, vue, contrôleur) introduite par Smalltalk-80 [MEV 87, GOL 83].

De même, la présentation du code de la figure 3.2 peut être une fenêtre de simulation assimilable à un débogueur évolué permettant de fournir la trace des informations du programme lors de son exécution. Aussi l'utilisation d'un langage orienté objet tel que Smalltalk pour implémenter notre système nous permet de réutiliser par héritage son débogueur en étendant la plupart de ses fonctionnalités.

III.3 - LE MODELE M.V.C. DE SMALLTALK ET LE CONCEPT DE DEPENDANCE

Le modèle M.V.C. repose sur la notion de dépendance qui permet de déclarer que les propriétés ou actions d'un objet sont directement tributaires des priorités ou actions d'un autre objet, sans avoir à définir le second comme variable d'instance du premier. La déclaration d'un lien de dépendance entre deux objets est réalisée par un envoi de message à l'un des objets, dont l'argument est l'objet dépendant. Un lien de dépendance est toujours orienté ; un objet peut posséder des dépendants sans pour cela être lui-même dépendant d'un autre objet. Chaque objet dépendant est informé des modifications survenues dans l'objet dont il dépend par un envoi de message de remise à jour (Cf. figure 3.5).

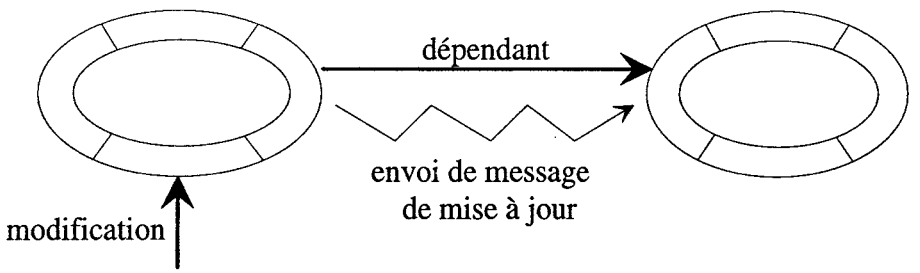


***Figure 3.5** : Relation de dépendance entre deux objets.*

L'objet possédant des dépendants décide lui-même quelles sont, parmi les modifications qu'il peut subir, celles dont ses dépendants doivent être informés. En résumé, un dépendant est un objet dont une partie de l'activité dépend de l'activité d'un autre objet.

L'utilisation la plus significative du mécanisme de dépendance concerne la construction de l'interface multi-fenêtre de Smalltalk qui repose sur trois entités, le modèle, la vue, et le contrôleur. L'ensemble est appelé modèle M.V.C. et permet d'obtenir une interface interactive (Cf. figure 3.6).

- Le modèle est l'objet contenant les informations que l'on veut afficher.

- La vue est une représentation qui se veut pertinente d'une des perpectives des informations du modèle jouant le rôle d'interface de sortie.

- Le contrôleur constitue l'interface d'entrée dont le rôle est de repérer les manipulations de l'utilisateur, en particulier de la souris et du clavier, et de lancer l'exécution des actions correspondantes. Comme ces actions se rapportent à l'information affichée, elles s'adressent généralement au modèle, parfois directement à la vue.

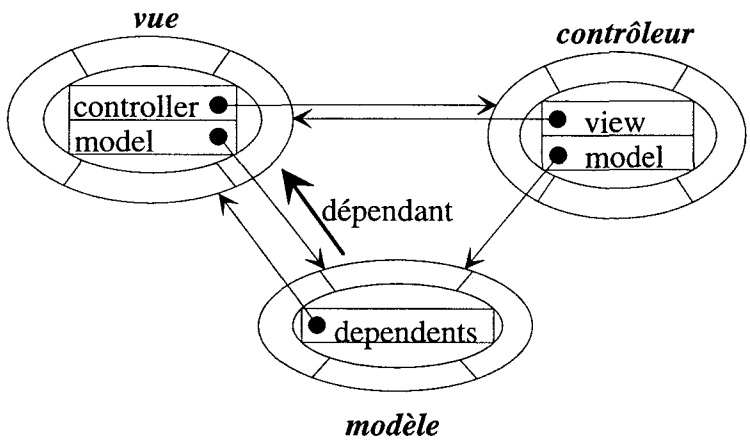


Figure 3.6 : *Relation d'instanciation et de dépendance du modèle M.V.C.*

Ce modèle d'architecture est ainsi utilisé par Smalltalk pour implémenter toutes les fenêtres de l'environnement de programmation, *"Browsers", "Transcript", "Inspector", "Debugger"*, etc.

Mais la fonctionnalité la plus importante du modèle M.V.C. dépend de la variable d'instance "dependents" qui est une collection comprenant toutes les vues dépendantes montrant le même modèle. Cette structure permet d'afficher et de mettre à jour automatiquement plusieurs représentations du même modèle dans différentes sous-vues d'une fenêtre.

III.4 - LE DEBOGUEUR DE SMALLTALK

Le débogueur de Smalltalk est un outil performant permettant la mise au point des programmes. En fait, il permet de visualiser les méthodes et les variables impliquées dans l'évaluation d'une expression pendant l'évaluation elle-même.

Originellement le débogueur de Smalltalk se décompose en quatre sous-vues (Cf. figure 3.7) :

- la première sous-vue est la pile des contextes qui présente, sous la forme d'une liste, les méthodes ou les messages appelées lors de l'exécution du programme. Grâce à cette sous-vue, il est possible de sélectionner un contexte antérieur à celui qui est actuellement présenté et de reprendre éventuellement l'exécution du programme à partir de celui-ci. Le débogueur se charge alors d'exécuter les différentes méthodes appelées en dépilant sa pile de contextes et initialise le message sélectionné.

- La seconde sous-vue de type texte présente la méthode en train de s'exécuter, avec sélection automatique de l'expression en cours d'évaluation déterminée par l'arbre syntaxique du programme (cf. Annexe A.4.2). Cet arbre est systématiquement reconstruit, par le débogueur, en appelant le compilateur à chaque appel de nouvelles méthodes pour initialiser son propre contexte.

- Enfin, nous avons la présentation de deux inspecteurs contenant respectivement les variables privées de l'objet pour l'une et, les paramètres de la méthode ainsi que ses variables temporaires pour l'autre. Ces deux inspecteurs sont eux-mêmes subdivisés en deux sous-vues qui comportent respectivement la liste des variables inspectées dont la sélection de l'une de celles-ci entraîne l'affichage du contenu de cette même variable dans la seconde sous-vue sous forme textuelle.

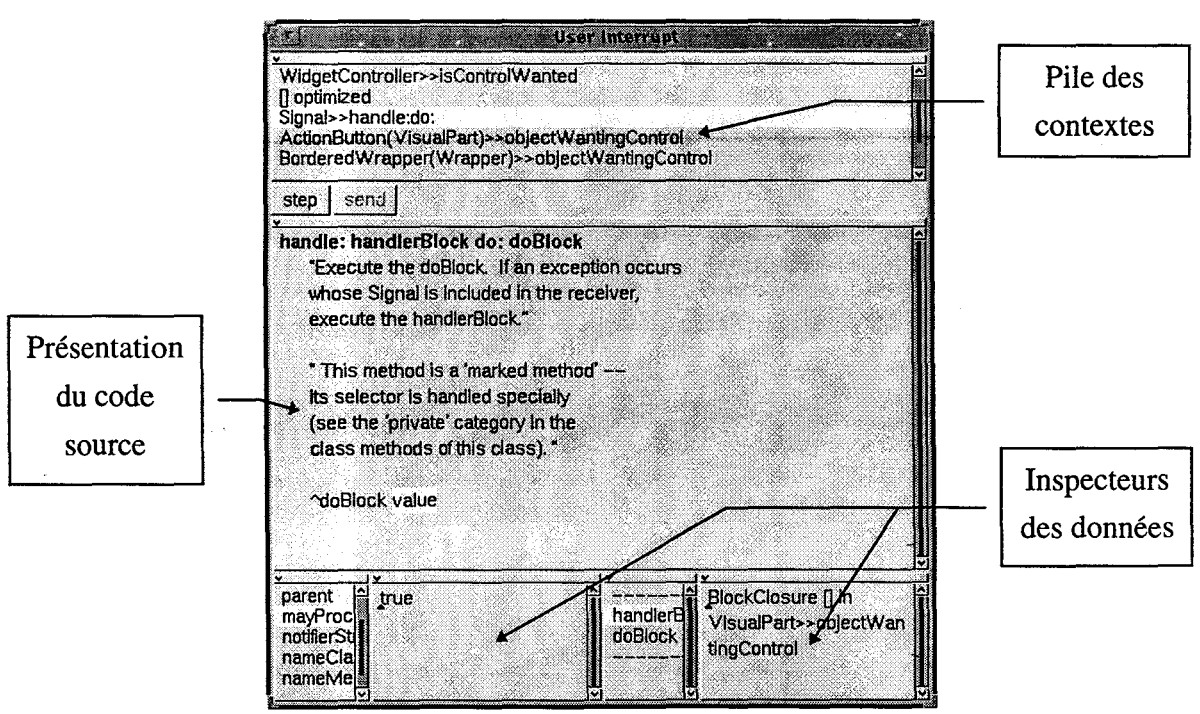


Figure 3.7 : Le débogueur de Smalltalk

D'autre part nous pouvons inspecter automatiquement chacune des données du programme dans des fenêtres autonomes en sélectionnant dans la liste présentée, par les inspecteurs du débogueur, les variables que l'on désire visualiser. Mais cette visualisation est une inspection de l'état de la variable à un instant donné car celle-ci n'est pas remise à jour au cours du déboguage du reste de l'application.

III.5 - GESTION DES EVENEMENTS DANS LE SYSTEME PROPOSE

Les événements dans l'animation d'algorithmes correspondent à une transformation d'état des informations du système visualisables par l'utilisateur.

La détection de ce changement est gérée par la simulation qui est chargée de signifier à ses différentes représentations d'effectuer, si nécessaire, la remise à jour de l'image entraînant par là même l'animation. Aussi, le changement d'état d'une variable va être un des signes d'un éventuel rafraîchissement graphique. Nous obtenons, de ce fait, une simulation à événements discrets consistant à sauter d'une date d'occurrence d'événement à un autre.

Comme nous l'avons vu précédemment, plusieurs techniques ont été employées pour générer l'animation par événements dans le cadre de l'animation d'algorithmes. Ainsi, certains systèmes portent sur le tracé des états du programme, d'autres portent sur les événements. Certains requièrent la modification du code source, d'autres pas (cf. chapitre II.4). Mais, l'interactivité de ces différents systèmes est directement liée aux différentes techniques employées.

Pour ne pas obliger l'utilisateur à redéfinir l'animation lorsqu'il apporte une modification au code source du programme, nous nous sommes contraints à obtenir une animation sans modifier ce code. Ainsi, pour réactualiser la valeur des différentes informations visualisées, la technique par déclaration qui consiste à regarder consécutivement la valeur des objets inspectés, et à chaque pas de la simulation, de comparer leurs valeurs actuelles et leurs valeurs enregistrées au pas précédent est particulièrement bien adapté à ce problème. Si les objets sont différents, la méthode consiste en fait à lancer automatiquement la remise à jour des données affichées. Ceci correspond à la définition d'un événement intéressant dans le système Balsa [BRO 84].

Précédemment, nous avons vu que le concept de dépendance et le modèle d'architecture d'interface M.V.C. de Smalltalk permettaient d'afficher et de remettre à jour différentes représentations d'un modèle dans des sous-vues d'une fenêtre.

Sur ce principe nous avons bâti notre propre structure d'interface qui part de la constatation que si un modèle peut avoir diverses représentations dans une seule fenêtre, celui-ci pourrait, également, avoir différentes représentations dans plusieurs fenêtres et ainsi bénéficier des avantages des vues multiples et du multi-fenêtrage.

III.5.1 - Utilisation singulière du modèle M.V.C.

Le modèle étant l'objet contenant les informations que l'on veut afficher sera, dans le système proposé, soit le contexte du programme en cours d'évaluation (c'est-à-dire la méthode ou la procédure traitée) soit un objet ayant une variable d'instance pointant sur celui-ci. Le contexte est, en fait, obtenu par l'intermédiaire de la fenêtre présentant l'exécution de l'algorithme.

Pour réaliser le lien de dépendance, il suffit de donner le même modèle que celui de la simulation à chacune des fenêtres d'information que va ouvrir l'utilisateur dont les relations de dépendance se limitant entre les vues et le modèle sont présentés en figure 3.8.

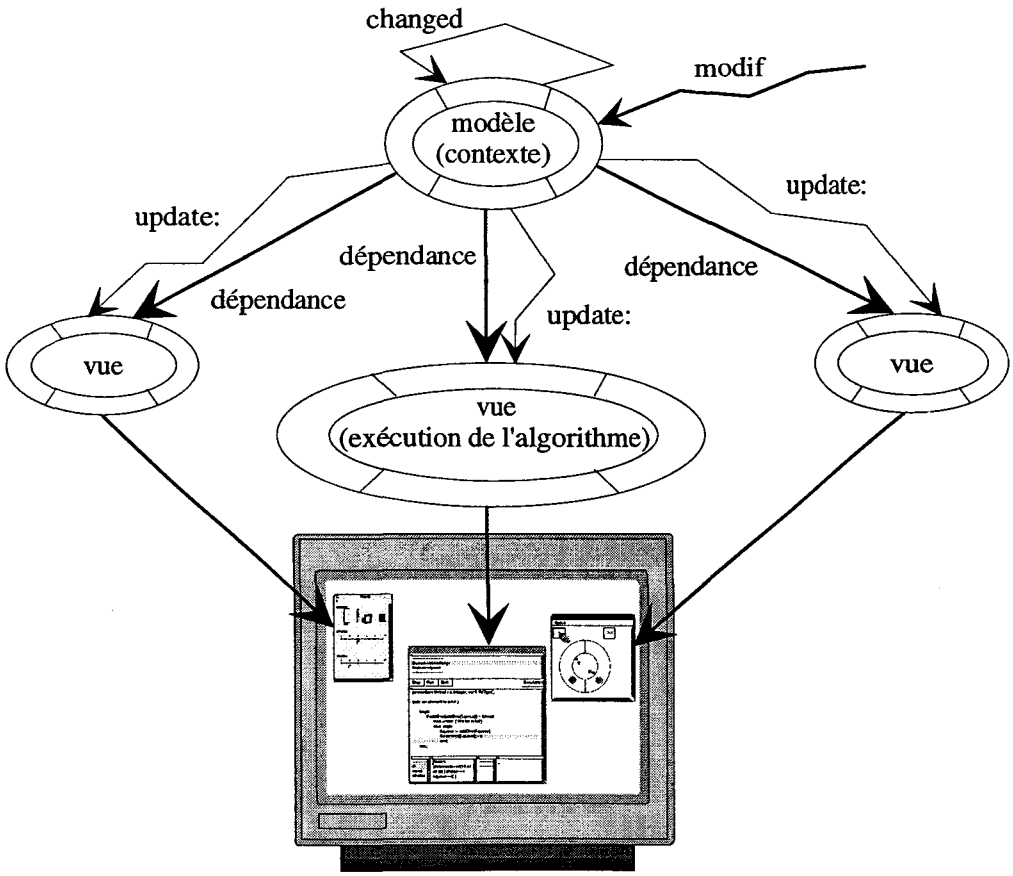


Figure 3.8 : *Schématisation des relations de dépendance*

Aussi, lorsque la modification du contexte est susceptible d'altérer l'information affichée par une fenêtre, le modèle s'envoie le message "changed". Cela a pour effet d'envoyer automatiquement le message "update:" [1] à toutes les fenêtres dépendantes par le mécanisme dit de dépendance. Lorsqu'elle reçoit ce message, la fenêtre accède à son modèle pour connaître son nouvel état et rafraîchit l'affichage si nécessaire.

Initialement, le message de remise à jour ("update:") est envoyé à toutes les fenêtres dépendantes qui regardent l'information du modèle qui le concerne avant de lancer, si nécessaire, les méthodes d'affichage et d'animation. Mais, ce type de mécanisme est dans la majorité des cas superflu, car l'information modifiée ne concerne qu'un nombre très limité de fenêtres dépendantes. Aussi, pour une plus grande rapidité d'animation nous détectons les variables altérées et nous n'adressons le message de remise à jour qu'aux fenêtres concernées par ce changement.

Cette relation de dépendance nous permet également de contrôler bien d'autres mécanismes. Par exemple, nous pouvons effectuer la fermeture automatique de toutes les fenêtres dépendantes du modèle et donc de la simulation. Ceci accroît la convivialité du système qui ferme automatiquement toutes les visualisations d'information obsolètes.

III.6 - UNE SIMULATION CONVIVIALE

Lors de la création de notre simulation, nous avons souhaité que l'accès aux différentes visualisations d'information d'un programme ne puisse être effectué que par l'intermédiaire de la présentation de l'exécution du code source. La visualisation de celui-ci a donc été déterminée comme l'élément central de notre système car il nous permet d'accéder à la visualisation graphique du comportement d'un programme et de ces données.

Comme nous l'avons introduit précédemment, la présentation de ce code est basée sur le débogueur de Smalltalk. Cette fenêtre conserve, en fait, essentiellement la même architecture de base. Les modifications apportées ne concernent principalement que la gestion interne des méthodes qui agissent sur la manière d'afficher les informations.

---

[1] Ce message comporte toujours un argument. Si le modèle s'envoie le message "changed" l'argument du message "update:" est l'objet "nil". Si le modèle s'envoie le message "changed:", l'objet passé en argument du message "update:" est le même que celui du message "changed:".

L'accès à cette fenêtre se fait comme lors d'un déboguage d'un programme à savoir en introduisant un point d'arrêt dans celui-ci. Concrètement, nous incorporons dans les algorithmes que nous voulons visualiser un message d'ouverture qui nous est propre. Celui-ci nous permet, ainsi, de contrôler la première instruction traitée et d'initialiser le contexte de la simulation (Cf. figure 3.9).

Mais ce message n'apparaîtra pas, à l'utilisateur, lors de l'exécution du programme grâce à un mécanisme complexe du compilateur qui nous permet de rendre différent le texte affiché et l'arbre syntaxique qui a généré le code intermédiaire lié à ce texte (cf. Annexe A.3.4.2).

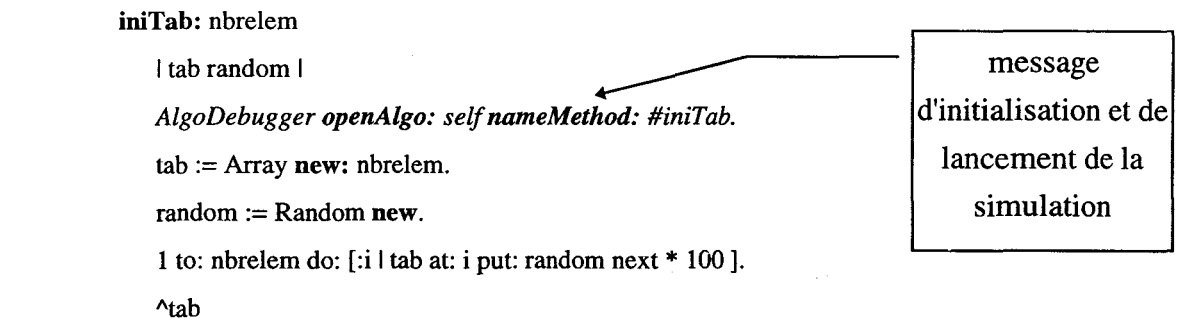


Figure 3.9 : *Exemple d'une introduction du message d'initialisation par l'analyseur syntaxique de Smalltalk ; dans un algorithme initialisant un tableau.*

Ce message ne sera introduit par l'analyseur syntaxique de Smalltalk, appelé "Parser", que si la méthode ou la procédure compilée appartient à une sous classe prédéfinie. Cette technique évite au compilateur de Smalltalk d'introduire ce message à toutes les méthodes qui pourraient être ultérieurement définies dans le système.

D'autre part, l'augmentation du code originel n'altère ni l'exécution, ni l'évaluation des performances de l'algorithme car ce message est traité avant l'ouverture effective de la fenêtre de simulation. Aussi, pour l'utilisateur la première instruction qui va être évaluée par notre système est le prochain message du programme (en l'occurrence le message "new:" pour l'exemple présenté ci-dessus). Par conséquent, pour celui-ci le message d'ouverture est considéré comme inexistant d'autant plus qu'il ne le visualise pas.

Ce message se charge, en fait, d'interrompre le processus en cours ; c'est-à-dire l'exécution de la méthode, d'initialiser le modèle et d'ouvrir la fenêtre de simulation (Cf. figure 3.10).

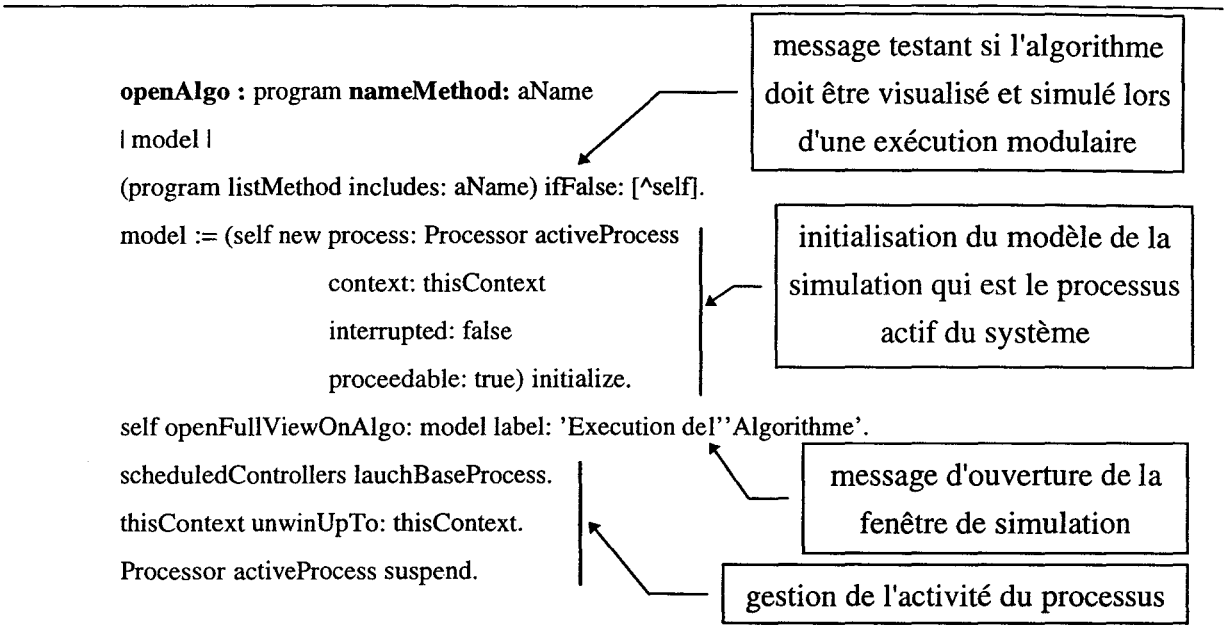


Figure 3.10 : Message de gestion d'initialisation de la simulation.

Nous filtrons, d'autre part, les messages lançant l'exécution de notre fenêtre de simulation car ils ne doivent pas apparaître dans la pile des contextes, comme ils le seraient dans le débogueur originel de Smalltalk que se soit lors de l'ouverture de cette fenêtre ou lorsque l'algorithme a fini d'être évalué.

Mais, les principales modifications du débogueur résident dans la modélisation des inspecteurs. Ceux-ci présentent respectivement, pour des programmes Pascal, les variables globales de la procédure en cours d'évaluation et ses variables locales. Par contre, pour des programmes Smalltalk, nous visualisons les variables privées qui ne font pas partie de la gestion de la simulation et les paramètres de la méthode exécutée ainsi que ses variables temporaires. Aussi, suivant le langage employé pour écrire nos algorithmes nous effectuons une initialisation différente de ces inspecteurs et nous ne présentons que les informations qui ont trait à l'algorithme exécuté.

D'autre part, par leur intermédiaire, l'utilisateur peut modifier, quel que soit le langage employé, la valeur des différentes variables utilisées avant de relancer une exécution.

Pour ce faire, il suffit de changer la valeur d'une donnée affichée textuellement par un des inspecteurs et de compiler le texte pour lui affecter.

Nous avons également ajouté différents boutons qui permettent à l'utilisateur d'accéder directement à certaines fonctionnalités de la simulation. Ainsi, la visualisation de synthèse est directement accessible par le bouton nommé "synthesis visualization". De même, l'utilisateur peut choisir très facilement deux modes de fonctionnement : le pas à pas et l'exécution continue par l'intermédiaire des boutons "Step" et "Run" (Cf. figure 3.11).

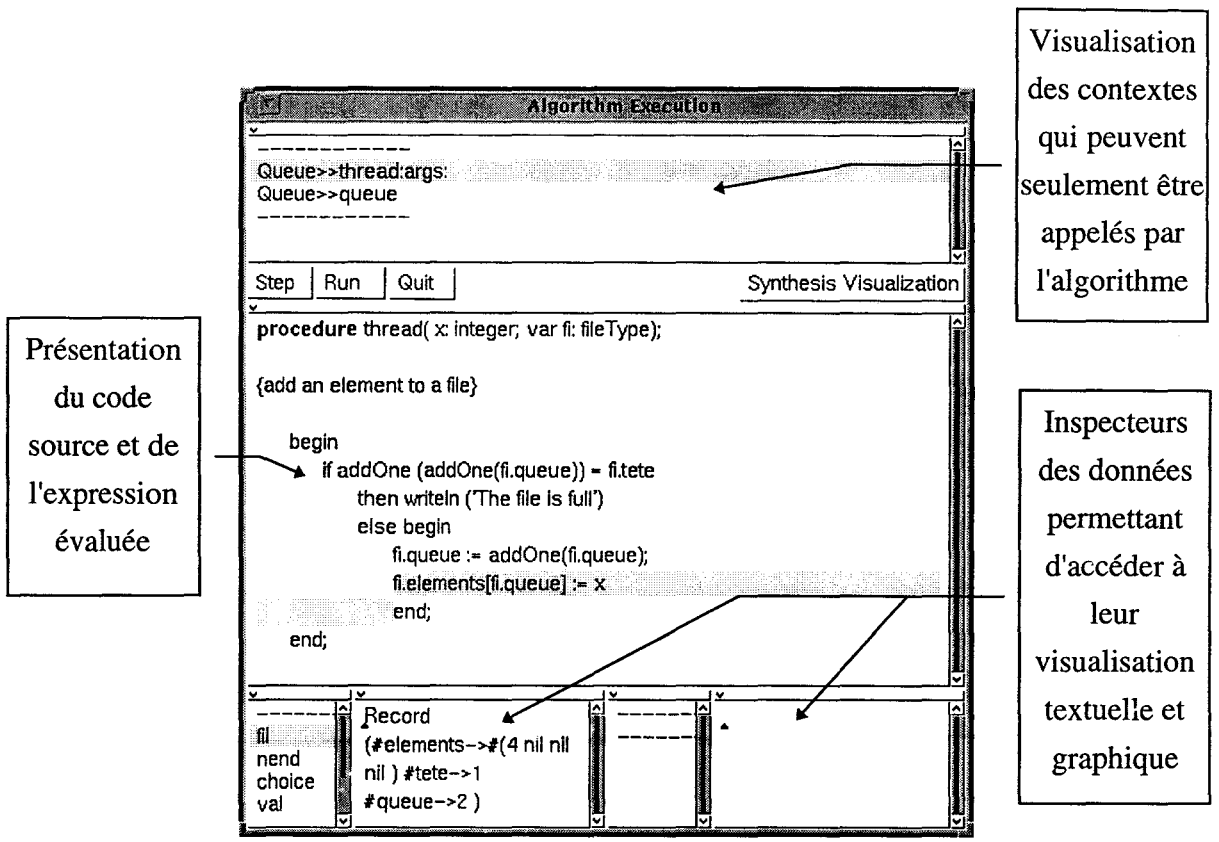


***Figure 3.11** : Présentation d'une procédure, de l'expression évaluée et d'une structure d'enregistrement (fil)*

Ces deux modes appellent automatiquement les sous-programmes de l'algorithme. Ainsi, le moteur gérant la simulation va tester, systématiquement et de manière transparente vis-à-vis de l'utilisateur, si l'expression évaluée et sélectionnée est un message ou une procédure du programme exécuté. Si ce test est validé la méthode ou la procédure est invoquée et affichée ; dans le cas contraire la méthode affichée reste celle qui est sélectionnée.

Ce type de fonctionnement, par défaut, n'est pas fondamentalement contraignant car l'utilisateur a la possibilité de choisir un contexte antérieur à celui qui lui est présenté par l'intermédiaire de la pile des contextes qui est construite sous la forme d'une liste. Il lui est

donc facile de ne pas visualiser le traitement de la méthode invoquée en sélectionnant un contexte antérieur et en l'exécutant. Ceci provoque également l'évaluation de tous les contextes postérieurs et leur dépilement.

Il est toutefois possible, par l'intermédiaire d'un menu fugitif associé à la simulation, de sélectionner d'autres modes de fonctionnement dont en particulier un mode de travail différent en pas à pas. En fait, celui-ci a été séparé en deux méthodes distinctes. La première est comparable au fonctionnement décrit précédemment, et la seconde évalue chaque expression sans jamais faire appel à un sous programme et, par conséquent, ne change pas de contexte avant que la méthode traitée ne soit définitivement exécutée.

Ainsi, notre simulation possède plusieurs fonctionnements possibles accessibles directement ou associés à un menu fugitif qui sont :

- Le pas à pas,
  - avec appel automatique à un sous programme,
  - sans appel à un sous programme,
  - avec appel explicite de l'utilisateur à un sous programme.

- L'exécution automatique,
  - de tout le programme avec sélection de l'expression évaluée, appel automatique à un sous programme et contrôle de la vitesse d'exécution,
  - d'une partie du programme en définissant un point d'arrêt à la souris dans le texte source,
    - avec sélection de l'expression en cours d'évaluation, ou
    - sans sélection de l'expression en cours d'évaluation.

D'autre part, l'exécution automatique est un processus ou une tâche lancé indépendamment de toute autre simulation. Ainsi nous pouvons exécuter plusieurs algorithmes en parallèle et comparer les différentes méthodologies de résolution pour un problème particulier.

III.7 - LA VISUALISATION DES DONNEES

Pour chacune des variables présentées par les inspecteurs de la simulation ou pour n'importe quelle expression retournant une donnée, définie par l'utilisateur ou sélectionnée à la

souris dans le programme, nous pouvons ouvrir une fenêtre nous permettant de visualiser celle-ci.

Ces fenêtres de visualisation de données sont subdivisées en deux sous-vues qui affichent respectivement le nom de la variable ou l'expression inspectée sous forme littérale et dans la seconde sous-vue la représentation de la valeur de l'objet concerné (Cf. figure 3.12). La visualisation s'effectue, alors, sous une forme graphique si la méthode d'affichage et d'animation est définie ou par défaut sous une forme littérale.

Une fonction zoom a été, également, associée à la vue graphique avec l'adjonction d'un ascenseur vertical et horizontal permettant de visualiser certains détails de l'affichage. Ceci se révèle utile lors de la visualisation d'objets constitués de plusieurs données qui ont entre elles des grandeurs très dissemblables ou qui contiennent un nombre de variables trop importantes pour avoir un affichage lisible. Cette fonction est accessible à la souris par un double click et permet de changer la taille de l'objet affiché avec un facteur d'échelle vertical et horizontal indépendant.

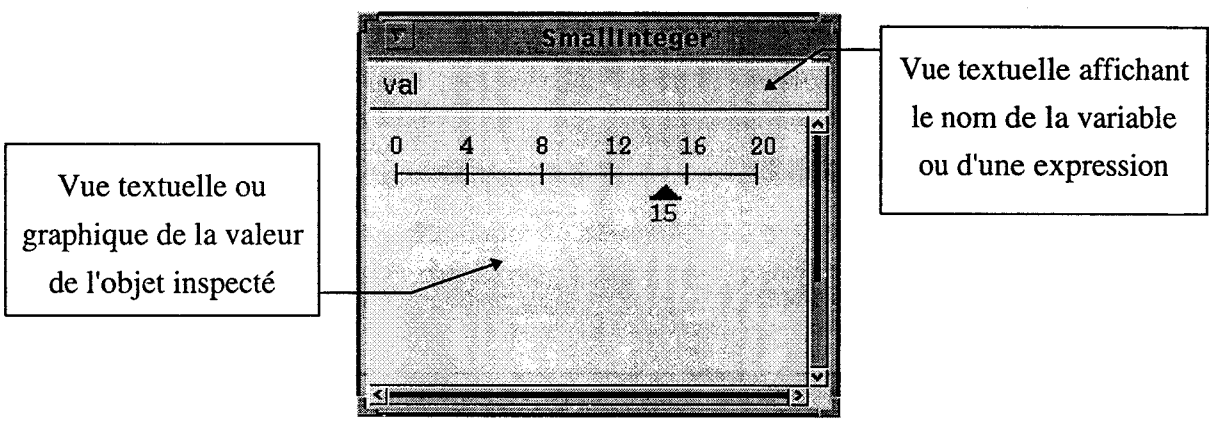


Figure 3.12 : Visualisation d'une donnée.

Pour lier une représentation particulière à un objet, on implante au niveau de la classe ou d'une de ses super-classes la définition de son animation, de la même manière que chaque objet connaît sa représentation textuelle dans l'environnement originel de Smalltalk avec le message "printOn:". Ainsi, la représentation et l'animation des données sont obtenues indépendamment de l'algorithme considéré puisque nous pouvons demander l'affichage de celles-ci, en invoquant toujours le même message et donc, sans connaître, a priori, la méthode de visualisation attribuée à chacune d'elles.

Pour remettre à jour l'affichage de ces différentes visualisations de données et lancer l'animation, ces fenêtres conservent un lien de dépendance, comme nous l'avons décrit précédemment, avec la fenêtre de simulation. Le système se charge, alors, de calculer les images intermédiaires pour effectuer l'animation. Celle-ci est ainsi constituée de transitions entre deux images graphiques dites agréables car visuellement l'animation paraît continue.

Mais, les besoins pour la représentation des données ne sont pas toujours les mêmes. En fait, ils dépendent de la structure de données visualisée et du nombre d'informations que nous allons pouvoir leur fournir. En effet, nous pouvons distinguer deux catégories de données :

- les structures de données de type simple tels que les entiers, les réels, les variables booléennes, etc.,

- les structures de données composées tels que les tableaux, les pointeurs, les enregistrements, etc.

En fait, la visualisation de données simples ayant une seule information à transmettre, c'est-à-dire leur propre état, ne nécessite pas de technique particulière à mettre en œuvre. Mais ce n'est pas toujours le cas pour des données composées car celles-ci ont besoin de véhiculer un nombre d'informations plus important pour être représentatif de leurs activités.

III.7.1 - Visualisation de données composées

Pour illustrer cette forme de visualisation, nous allons prendre un exemple simple de simulation, en l'occurrence, un algorithme de tri employant la structure de donnée du type tableau. Nous avons ainsi, quelle que soit la méthode de traitement choisie, à un instant donné, un échange entre deux éléments de ce tableau. Cet échange, informatiquement, ne peut être réalisé qu'en trois étapes et faisant intervenir une variable temporaire (t := a[i]; a[i] := a[j]; a[j] := t). Si nous ne contrôlons que l'état du tableau nous ne pourrons associer une animation qu'aux deux dernières affectations. Mais si nous donnons au tableau une information supplémentaire correspondant à l'état de la variable "t" nous pouvons contrôler et visualiser les trois affectations qui ont lieu lors de cet échange.

Pour fournir cette information supplémentaire, la technique mise en œuvre consiste à lier, par l'intermédiaire d'une variable d'instance de la donnée composée, l'objet constituant cette information (Cf. figure 3.13). Ce lien peut être ajouté ou enlevé au cours de la simulation

par l'utilisateur, qui choisit par l'intermédiaire d'un menu fugitif parmi toutes les variables de l'algorithme celle qu'il estime représentative de l'activité visualisée. Ce menu ne sera, en fait, accessible que pour des données composées préétablies.

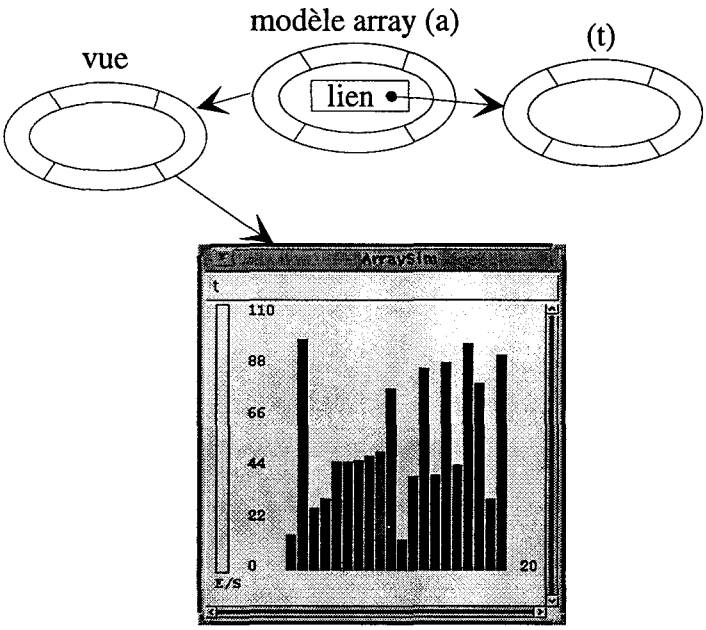


Figure 3.13 : *Le tableau nommé "a" a une variable d'instance qui pointe ou non sur une donnée nommée "t" qui permet de visualiser lors d'un tri cette variable temporaire.*

Il est également possible de visualiser, toujours pour le même exemple, les index (i, j) du tableau qui représentent les deux éléments pointés lors de l'interprétation du programme.

Ceci est obtenu en employant une autre technique qui consiste à piéger les messages d'affectation et de consultation de cette donnée (par exemple, "at:" et "at:put:", pour un tableau)[2]. Les deux index ainsi obtenus sont ajoutés au tableau comme deux informations supplémentaires, ce qui nous permet de lancer automatiquement la remise à jour de la représentation et de visualiser les éléments pointés du tableau. Ces informations nous permettent, également, de vérifier la cohérence entre la valeur de la donnée correspondant à l'index "i" ou "j" et la valeur de l'objet de la variable "a" ou "t" réaffectée.

(2) Cette technique a été plus systématiquement utilisée dans Animus [DUI 86a] qui piège certains sélecteurs afin qu'ils déclenchent une série d'instructions graphiques.

La remise à jour de l'affichage ne sera plus simplement effectuée en scrutant la donnée visualisée, mais en tenant compte également de toutes les informations qui lui sont associées.

III.8 - VISUALISATION ABSTRAITE OU DE SYNTHESE

Lors de la visualisation des différentes interactions de l'algorithme avec le tableau, nous pouvons remarquer que nous avons construit une animation plus abstraite en lui fournissant des informations supplémentaires. Pour généraliser ce type de visualisation, il suffit de ne plus considérer les informations comme se rajoutant à une donnée, mais en fait qu'une information est constituée de plusieurs données. Ainsi, c'est la combinaison de ces données qui va créer une représentation particulière.

Pour réaliser ce type de visualisation, nous créons un modèle indépendant qui comporte toutes les données nécessaires à son affichage. Pour ce faire, ce modèle a, par exemple, une variable d'instance nommée "dicLien", qui est un dictionnaire, contenant toutes les structures de données requises. Ainsi, ce dictionnaire a pour clé le sélecteur d'une des variables de l'animation et pour valeur la donnée de l'algorithme associée sous sa forme textuelle, qui pointe elle même sur la valeur de l'objet considéré. Ce type de structure nous permet de rendre le modèle et la visualisation indépendants de l'algorithme car nous pouvons associer à une clé du dictionnaire n'importe quelle valeur et donc n'importe quelle structure de données.

Pour mettre à jour cette représentation, nous adoptons toujours le même principe en balayant à chaque pas de simulation le dictionnaire pour comparer la valeur des nouvelles données pointées par celui-ci et celles qui ont été mémorisées au pas précédent et activer ou non la simulation.

Nous mettons l'accent sur le fait que cette représentation est un objet qui ne dépend pas de l'algorithme ; ainsi nous pouvons réutiliser son implémentation pour un autre programme. Il suffit, pour ce faire, de lier aux clés du dictionnaire une autre structure de données.

D'autre part, ces liaisons ne s'effectuent pas simplement entre une variable de l'algorithme et un objet graphique mais peuvent être également obtenues avec une expression quelconque retournant une donnée.

Nous montrons en figure 3.14 les relations que nous avons utilisées lors de l'animation abstraite d'une FIFO ("first-in first-out" premier entré, premier sorti) (3). Le modèle obtenu peut être associé aussi bien à un algorithme faisant appel à une donnée du type enregistrement à trois champs ou à trois données distinctes du type entier et tableau.

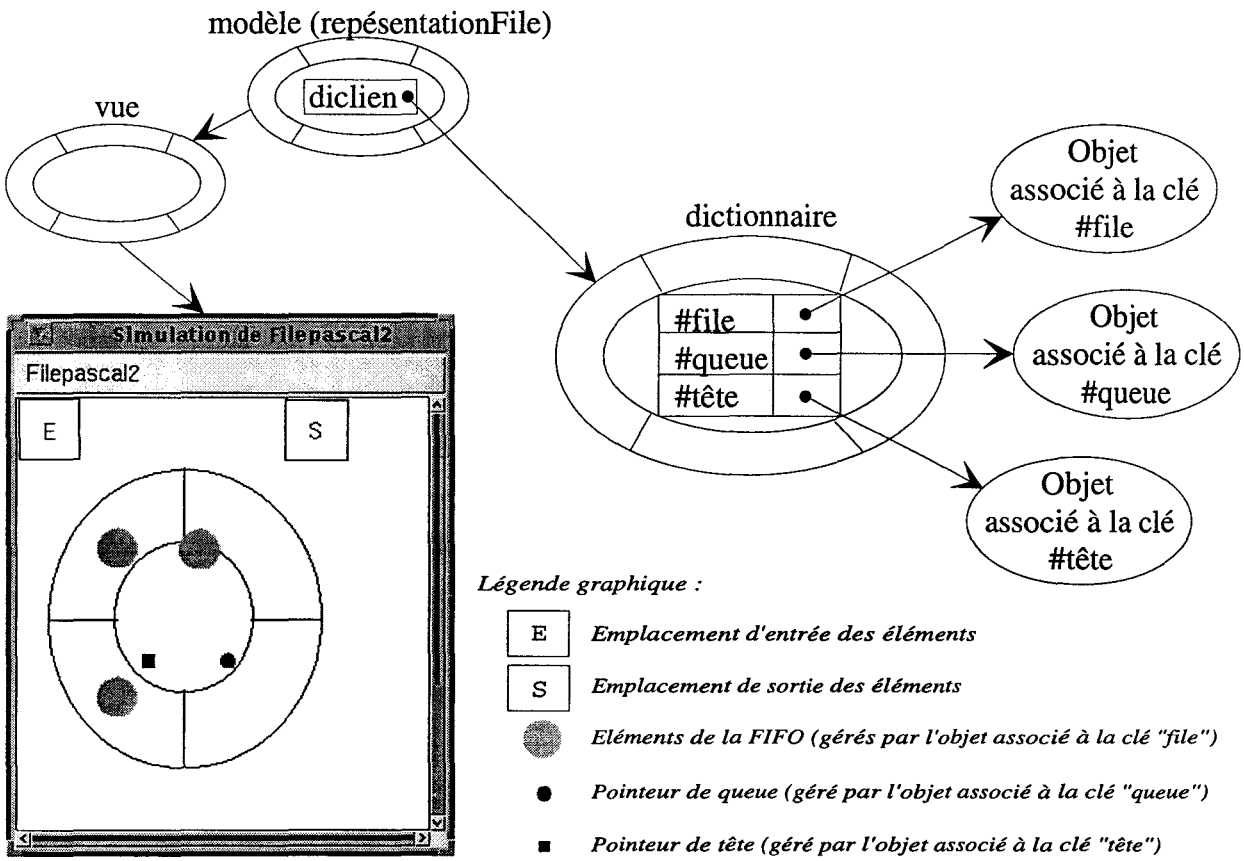


***Figure 3.14 :** Le modèle contient la méthode d'affichage de la vue et a pour variable d'instance un dictionnaire qui sert d'interface entre les données du programme et celles qui gèrent l'animation proprement dite.*

Ce type de visualisation peut nous permettre de montrer également de manière différente l'activité d'une donnée composée.

Ainsi si nous reprenons l'exemple d'un algorithme de tri sur un tableau ; nous pouvons montrer l'activité des deux index (i, j) ainsi que la variable temporaire (t) du

(3) La FIFO est représentée sous la forme d'un anneau circulaire découpé en plusieurs cellules dont la dernière est directement suivie par la première. Cette représentation permet de montrer que l'on considère la file comme un ensemble de cellules continues et dénombrées mais n'ayant plus de début ni de fin. De même, elle permet de voir les cellules occupées et l'ordre de stockage et de déstockage des éléments.

programme en liant à la représentation ces différentes données. Nous perdons, toutefois, une certaine souplesse par rapport à la visualisation d'une donnée composée car ce n'est plus l'utilisateur qui choisit de lier la variable temporaire à l'animation.

D'autre part, nous n'obtenons plus simplement la représentation de l'activité d'une donnée mais bien une représentation de synthèse de l'algorithme. Cette différence se caractérise par une animation sensiblement différente car dans la première technique de visualisation la valeur d'un des index peut être changé sans que l'animation ne soit effectuée. Ceci est dû au fait que les données du tableau ne sont éventuellement modifiées que lorsque l'évaluation d'une expression de l'algorithme utilise, pour la donnée considérée, le message "at:" ou "at:put:". Par contre dans la seconde technique, l'animation est effectuée, dès que la valeur d'un des deux index change.

Si le résultat final semble être identique, ces deux animations correspondent bien à deux types de visualisation. L'une concerne l'activité d'une donnée dont l'affichage des index constitue une information et l'autre restitue l'activité de l'algorithme.

Il est toutefois préférable, pour un utilisateur naïf (Cf. chapitre II.2), de visualiser l'activité de l'algorithme car il pourra facilement associer l'évaluation des expressions à l'animation visualisée. Mais pour une personne plus expérimentée, le deuxième type d'animation de données peut être mieux adapté à son attente car son obtention est immédiate et moins dépendante du programme étudié.

III.9 - VISUALISATION DES PERFORMANCES D'UN ALGORITHME

Pour comparer les performances de différents algorithmes (le tri, par bulle, par insertion, etc.), il est possible d'ouvrir une fenêtre autonome présentant le nombre d'accès d'un programme, à la mémoire, que ce soit en lecture ou écriture, le nombre d'opérations élémentaires effectuées du type +,-, ..., le nombre d'affections, etc. (Cf. figure 3.15).

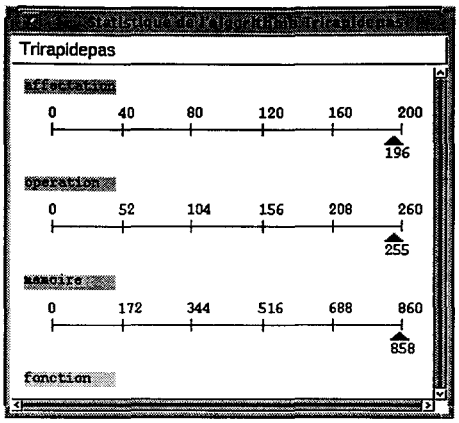


***Figure 3.15** : Fenêtre d'évaluation des performances d'un algorithme.*

Cette fenêtre peut être obtenue directement par l'intermédiaire de la fenêtre de simulation au cours de l'exécution d'un programme et sa remise à jour est effectuée à chaque pas de simulation. Celle-ci est gérée par l'entremise du concept de dépendance présenté au paragraphe III.5 et du piégeage de certains sélecteurs (Cf.paragraphe III.7.1).

L'utilisateur n'a plus qu'à affecter, pour un ordinateur donné, chaque fonction d'un poids correspondant au temps de traitement de celles-ci et calculer le temps global que mettent les programmes à être exécutés et ainsi mesurer la complexité algorithmique (en temps) de ceux-ci [BRA 87] suivant différents paramètres (type de processeur utilisé, fréquence de travail, etc.).

III.10 - VISUALISATION DU TAS

Nous pouvons ouvrir, par l'intermédiaire du menu fugitif associé à la fenêtre de simulation, une fenêtre visualisant le tas et les interactions de l'algorithme avec celui-ci (Cf. figure 3.16). Cette présentation peut aider un utilisateur à comprendre comment une expression est évaluée et communique avec la mémoire. Sa gestion s'effectue de la même manière que pour visualiser les performances d'un algorithme en mettant à jour la représentation à chaque pas de simulation par l'entremise du concept de dépendance et du piégeage de certains sélecteurs.

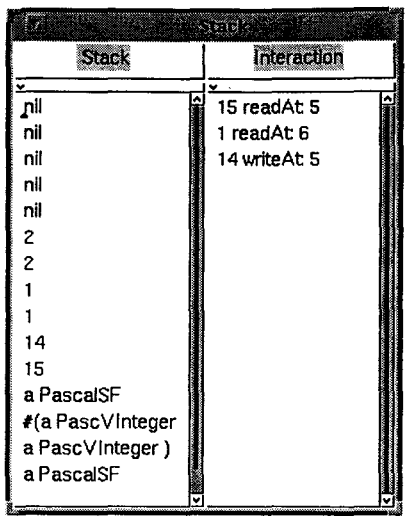


***Figure 3.16** : Fenêtre présentant le tas ainsi que des accès à celui-ci au cours d'un pas de simulation.*

La visualisation du tas reste, pour des expressions complexes, très difficile à interpréter car une même expression peut accéder une dizaine de fois à celui-ci. Ceci nous oblige, soit à présenter les états intermédiaires et par conséquent à ralentir la simulation, soit à présenter, comme nous l'avons fait, les interactions et l'état du tas après un pas de simulation puis laisser l'utilisateur interpréter l'information présentée.

Ce type de visualisation ne peut donc permettre d'expliquer le fonctionnement d'un tas mais sert plutôt à renseigner un utilisateur expérimenté sur la nature des interactions qu'effectue l'ordinateur avec le tas pour gérer ces données au cours de l'évaluation d'une expression.

III.11 - ARCHITECTURE D'UN MODELE GRAPHIQUE

Pour obtenir un système homogène et convivial, nous avons voulu avoir la possibilité d'élaborer les animations graphiques (de données ou de synthèse) de manière automatique; c'est-à-dire, sans être obligé de connaître les messages et les classes gérant le graphisme de Smalltalk. Nous avons pour se faire élaboré un modèle d'architecture graphique qui répond à tous les besoins en animation et à toutes les formes de visualisation.

III.11.1 - Les besoins en animation

L'animation d'algorithmes s'effectue par l'intermédiaire d'objets graphiques simples tels que la ligne, le rectangle, le cercle, le texte, etc. Tous les objets graphiques peuvent être utilisés pour composer un objet graphique plus complexe. Les méthodes d'animation sont tributaires des paramètres qui peuvent être attribués à ces objets.

Ainsi nous pouvons agir sur la couleur de l'objet, sur sa forme (par rotation ou par déformation suivant les deux axes), sur sa position (par déplacement), etc.

Lors de la manipulation d'algorithme, le besoin en animation n'est pas toujours le même. Ceci dépend de la structure de données et du changement d'état à représenter. En effet, nous pouvons étudier les méthodes d'animation suivant les deux catégories de données vues précédemment (simples et composées).

L'animation des structures de données simples fait appel à des méthodes d'animation dites élémentaires et qui sont énumérées ci-dessus. Concrètement, ces méthodes sont ramenées à une simple transcription de l'état de la variable représentée à un état du paramètre graphique (couleur, taille, valeur, etc.).

Par contre, l'animation des structures de données composées fait appel à des méthodes d'animation qui lui sont intrinsèques car liées aux types d'opérations que l'on peut effectuer sur elles. Ceci suppose donc de nouveaux besoins en animation.

En effet, il ne s'agit plus de représenter un état ou un changement d'état au sens élémentaire mais un comportement. Nous devons donc faire appel à des méthodes d'animation graphique permettant de simuler, par exemple pour un tableau, la création ou la suppression d'éléments, la permutation de deux éléments, etc.

Dans ce cas de figure, la simple transcription utilisée, se limitant aux changements des paramètres graphiques, donne une animation discrète et souvent difficile à comprendre. C'est la raison pour laquelle nous avons développé de nouvelles méthodes d'animation composée. Elles sont basées sur le même principe que les méthodes élémentaires mais prennent, de plus, en considération le changement d'état discret des données représentées. De ce fait, la transition d'un état initial à un état final est traduite graphiquement par une évolution continue et permet à l'utilisateur de mieux interpréter l'abstraction visuelle en lui permettant de suivre les changements sans à-coup.

Enfin, l'animation de synthèse fait appel aux deux types d'animation précédemment décrits puisqu'ils enveloppent aussi bien des données simples que des données composées.

Afin de couvrir la majorité des besoins, nous avons développé un modèle d'architecture générique applicable aux différents niveaux de présentation et types d'animation (Cf. chapitre II), baptisé P.Os.T (Présentation, Objet de Simulation, Traducteur) qui nous permet de créer à partir d'une interface conviviale une animation.

III.11.2 - Le Modèle P.Os.T.

Le modèle P.Os.T. a été originellement développé dans le cadre d'une simulation de processus industriel par M. Mostefai [MOS 93, 94a]. Ce modèle a été préféré aux modèles M.V.C., M.V.C. étendu ou P.A.C. car ces modèles présentent des inconvénients majeurs pour établir une animation (Cf. Annexe C).

Le modèle P.Os.T. dont l'architecture est présenteé en figure 3.17 et est composeé de trois éléments principaux :

- L'objet de simulation qui décrit la partie abstraite de l'application. Celle-ci peut être constituée d'une partie statique décrite pour un ou plusieurs attributs et d'une partie dynamique qui décrit son comportement. Pour la réalisation de notre animation, nous n'utilisons que la partie statique dont les attributs représentent les variables de l'algorithme.

- La présentation, chargée de l'aspect visuel de l'abstraction et assure l'affichage et l'animation de l'objet de simulation. C'est la composition d'un ensemble de composants graphiques simples tels que des rectangles, des cercles, des textes, etc.

- Le traducteur prévu pour assurer la cohérence des domaines abstraits et graphiques grâce à une bibliothèque de méthodes de traduction extensible par l'opérateur. Chacune de ces méthodes agit sur un des attributs des objets graphiques (couleur, position, rotation, déformation en suivant les deux axes, etc.).

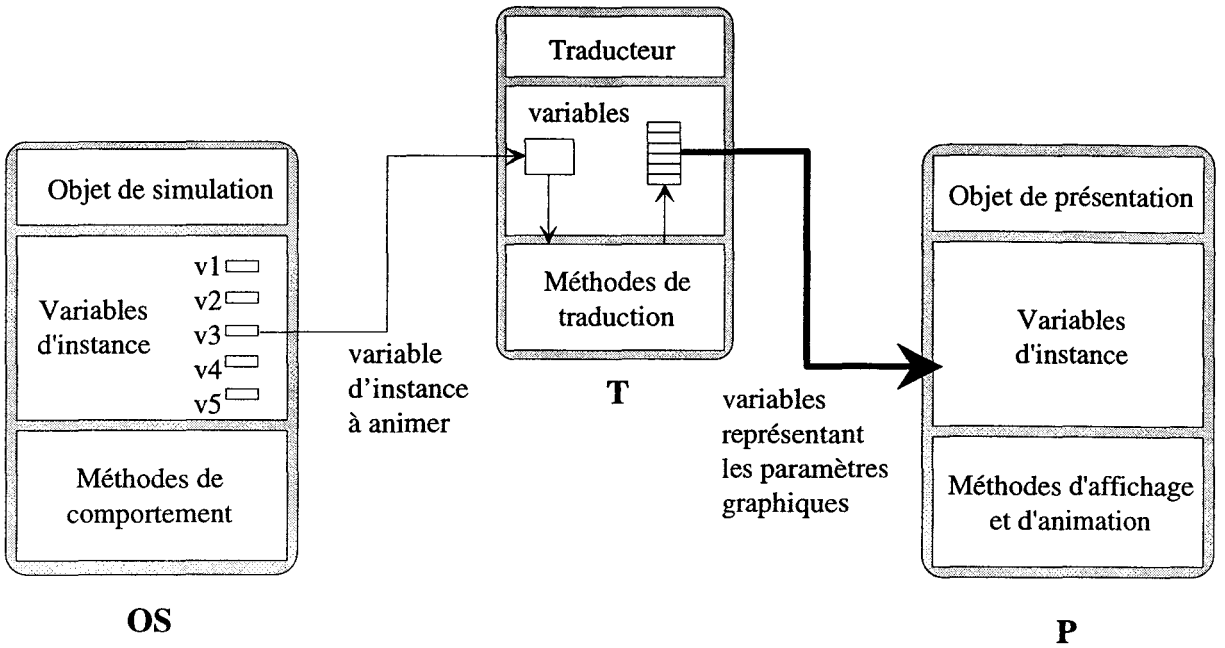


***Figure 3.17** : Architecture du modèle P.Os.T.*

III.9.11.1 - Composition graphique

La composition des objets graphiques est indispensable pour une représentation plus parlante de l'état des objets de simulation. Elle consiste à représenter un objet par une composition d'éléments graphiques simples de types différents : rectangles, cercles, textes, etc (Cf. figure 3.18).

Cette technique permet, entre autre, une meilleure représentation graphique des différentes informations de la simulation par l'adjonction, par exemple, de plusieurs commentaires textuels et de symboles graphiques explicites de l'objet représenté.

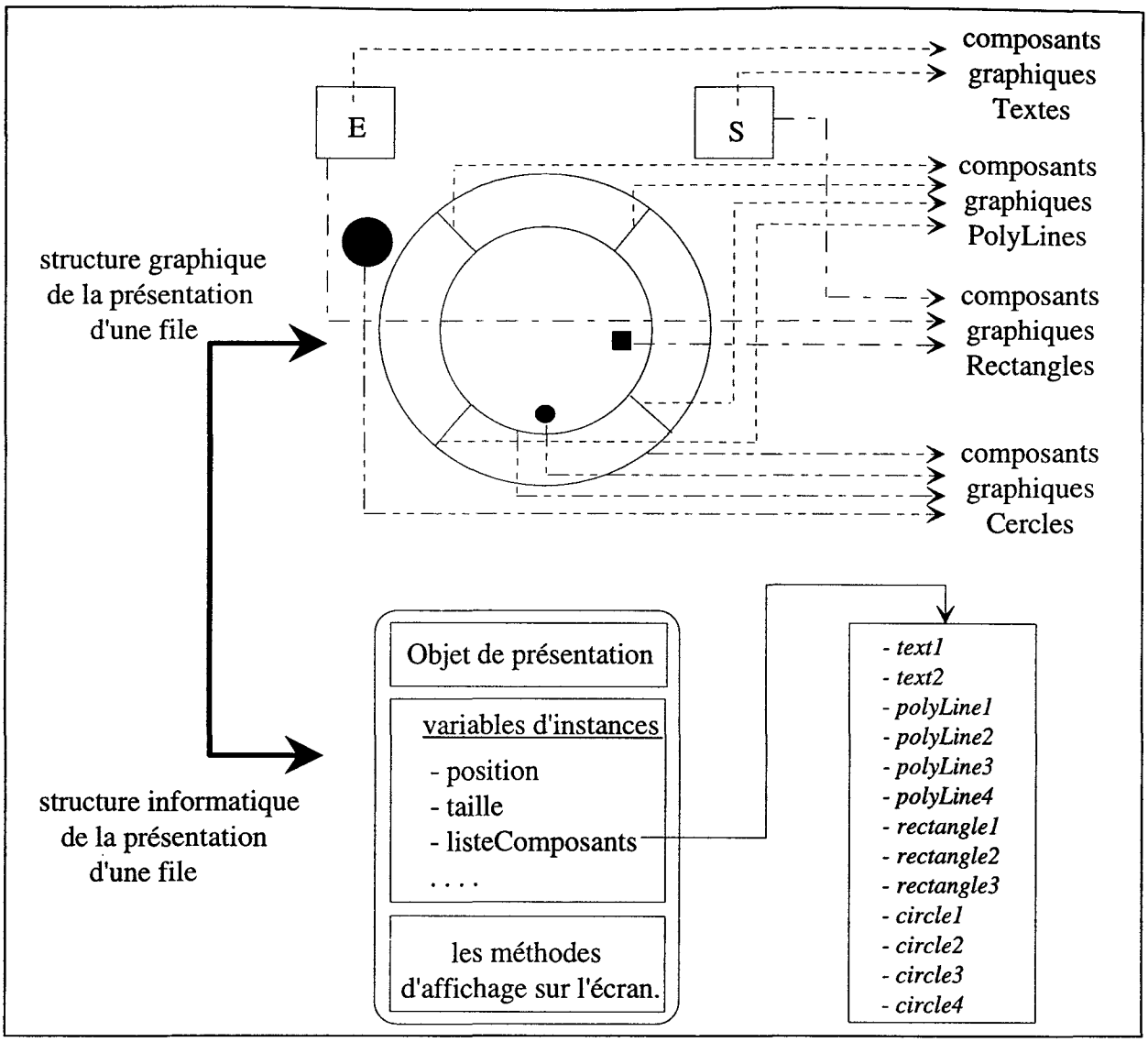


Figure 3.18 : *Composition et structure graphique de la FIFO de la figure 3.15*

La structure informatique de la présentation et sa gestion deviennent, également, plus complexes. En effet, l'objet composé qui décrit la représentation de l'information visualisée est constitué d'un ensemble de variables qui lui sont propres, correspondant aux paramètres d'affichage dans la fenêtre de visualisation : position, largeur, hauteur, facteur d'échelle, etc. Ce même objet fait référence, à l'aide d'une autre variable, à la liste des composants graphiques élémentaires le constituant ayant chacun leurs propres paramètres d'affichage : position (relative à la présentation), largeur, hauteur, couleur, etc.

Lors de l'affichage d'un objet composé, nous faisons appel à l'ensemble des composants graphiques de la liste, pour afficher un à un ceux-ci, en tenant compte d'une part de la position de référence de l'objet de représentation globale, ainsi que du facteur d'échelle qui lui est associé.

III.9.11.2 - Outils de construction du modèle P.Os.T.

La construction d'un modèle P.Os.T. se fait de manière interactive et en plusieurs étapes.

♦ La première consiste à créer l'objet de simulation, en utilisant le générateur de classes du sytème.

L'objet de simulation contient la méthode qui relie un nom de variable et sa donnée à une donnée du modèle graphique. Celui-ci peut ainsi être assimilé au modèle présenté précédemment pour la visualisation de synthèse car il aura dans l'animation d'algorithme, la même fonctionnalité.

Par conséquent, la notion d'objet de simulation indépendant est très importante, même si sa description est rudimentaire, car elle permet d'attacher à plusieurs algorithmes, qui ont le même comportement global, le même modèle P.Os.T (Par exemple, dans le cas de différents tris : rapide, par sélection, par insertion, etc.).

♦ La seconde étape consiste à créer l'objet de présentation en se servant d'un éditeur d'image graphique vectorisé. Cet outil permet à l'animateur de construire, sans aucune programmation, des présentations simples ou complexes. Il contient un ensemble de boutons que nous avons associé aux différents types d'éléments graphiques existants.

L'opérateur peut à volonté choisir ses propres paramètres d'affichages (couleur, largeur de trait, etc.) par l'intermédiaire des menus et des boutons de réglage mis à sa disposition (Cf. figure 3.19).

Enfin, dès lors que l'utilisateur a fini de construire l'objet de présentation du modèle P.Os.T, il peut le sauvegarder dans une bibliothèque de fichiers, afin de le récupérer, soit pour le modifier, soit pour l'affecter à un objet de simulation lors de la troisième étape.

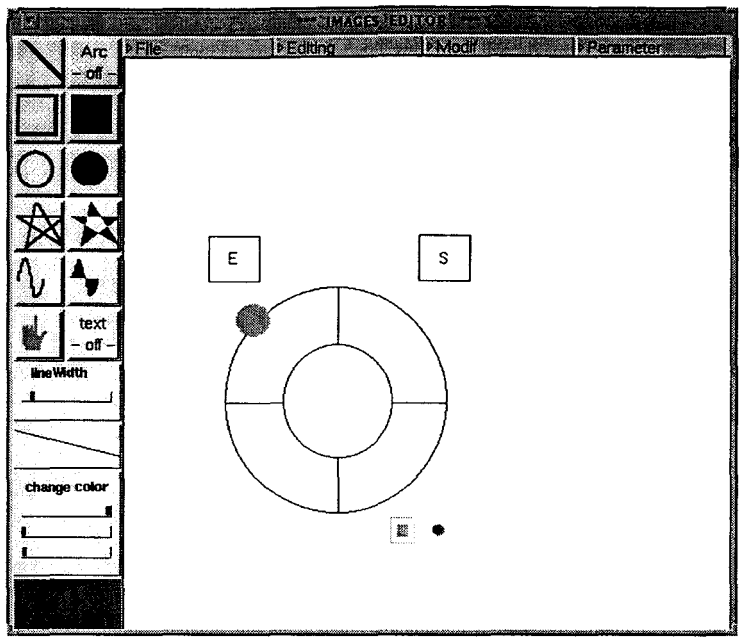


Figure 3.19 : *Editeur d'images.*

♦ Cette troisième étape, appelée phase de prototypage [4], consiste à relier les deux entités créées précédemment grâce à un objet traducteur et de choisir les animations souhaitées. Ce choix est fait en associant interactivement, par l'intermédiaire de la souris, un objet graphique à une méthode de traduction parmi les méthodes de la bibliothèque d'animation (Cf. figure 3.20).

L'objet P.Os.T. ainsi créé peut être également sauvegardé sous forme de fichier pour ne pas encombrer l'interface d'objets graphiques inutilisés.

[4] Cette phase est appelée ainsi en terme de langage objet, car nous créons un objet prototype P.Os.T instance de sa classe objet de simulation. Ce prototype constitue un modèle que nous pourrons copier et par la même lui affecter des variables d'instances particulières pour différentes animations [MOS 94a].

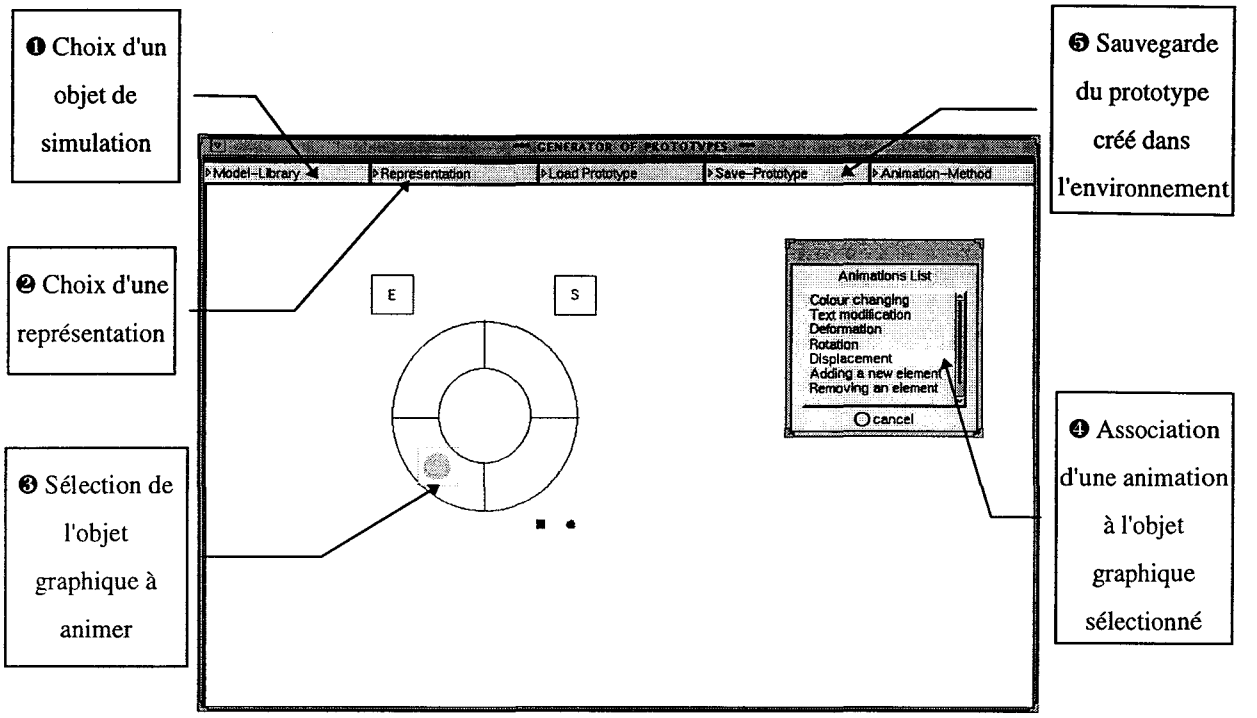


Figure 3.20 : *Exemple d'une phase de prototypage pour l'animation d'un algorithme FIFO*

III.11.3 - Adaptation du modèle P.Os.T. à l'animation d'algorithmes

A chaque animation nous associons un modèle P.Os.T. que nous avons adapté à l'animation d'algorithme, mais nous ne touchons pas au modèle global de l'architecture ; en fait, nous n'agissons pratiquement que sur les possibilités du traducteur [MOS 94b].

Ceci se caractérise, d'une part, par un enrichissement et une adaptation de la bibliothèque d'animation pour répondre aux besoins d'animation qui ont été énoncés précédemment, et d'autre part, par un accroissement des possibilités d'animations multiples.

Ainsi, si nous considérons les traitements effectués sur une structure de données composée, par exemple sur une variable du type collection, nous pouvons lui associer plusieurs comportements possibles, et ainsi plusieurs animations différentes. Nous pourrions avoir l'ajout, le changement de valeur ou la suppression d'un des éléments de cette collection dont chacune de ces actions correspond à une animation particulière ou à une combinaison d'animations particulières.

Le modèle P.Os.T., originellement, peut associer plusieurs représentations à une même variable en définissant des contraintes différentes, par exemple pour visualiser la valeur d'une variable sous forme alphanumérique et graphique. On ne peut toutefois avoir qu'une

seule méthode d'animation pour un élément graphique. Nous avons donc étendu le modèle de traduction en donnant la possibilité d'associer au traducteur plusieurs animations pour une seule variable de l'objet de simulation.

Ces adaptations nous ont permis de joindre à n'importe quel type de structure de données, simple ou composée, ainsi qu'à chaque représentation de synthèse d'un algorithme, un modèle P.Os.T. Cette association s'effectue, plus spécifiquement, par l'intermédiaire de l'objet de simulation.

Nous pouvons considérer, selon le type de structure de données visualisée et les besoins de l'utilisateur, trois formes d'animations :

- L'animation de structures de données simples (entiers, réels, variables booléennes ...) est assurée en associant un modèle P.Os.T. à chacune des variables animées. Une liste des méthodes de traduction élémentaire a été créée et stockée dans une librairie extensible correspondant aux animations standards telles que le changement de couleur, la modification du texte, la déformation rectangulaire, etc.

- L'animation des structures de données composées (collections, tableaux, pointeurs ...) est réalisée avec le même principe en utilisant un modèle P.Os.T. et une ou plusieurs méthodes de traduction. La particularité de ce type de structure de données est de présenter de nouveaux types d'opérations tels que : ajouter un nouvel élément à une collection, détruire un autre élément, etc.

- L'animation de synthèse est réalisée en associant à l'algorithme un modèle P.Os.T. qui est une visualisation permettant de décrire la métaphore du monde réel sous la forme d'une image représentative du problème traité. Cette représentation fait alors appel à un ensemble de variables du programme dont nous pouvons visualiser les interactions. Les nouvelles fonctionnalités du traducteur et les méthodes d'animation définies dans les deux catégories précédentes suffisent à animer une telle image.

De plus, lors de la phase de prototypage, l'utilisateur peut associer à chacune des méthodes d'animation un certain nombre de contraintes. Ainsi, un déplacement peut être dû à un parcours imposé, un changement de forme peut être relatif à une valeur maximum, etc. Ces différentes contraintes sont associées à la méthode d'animation comme un paramètre et sont traitées par celle-ci comme de simples conditions à résoudre.

Ces contraintes ne sont pas à confondre avec celles de l'objet de simulation qui sont chargées de lancer l'animation pour répondre au comportement de celui-ci. Par exemple,

l'objet de simulation peut être une cuve dont la contrainte est de vérifier le débit d'entrée et de sortie pour déterminer son niveau et de lancer l'animation pour que sa représentation reste fidèle.

Le son a été également expérimenté et constitue un paramètre de visualisation comme la couleur ou la position de l'image. Mais son utilisation reste restreinte car elle peut constituer, si elle est employée trop souvent, une gène pour l'entourage de l'utilisateur.

III.11.4 - Fonctionnalité de l'objet P.Os.T.

Originellement, le traducteur définit autant de contraintes que de variables à animer. Chaque contrainte correspond à une méthode de traduction qui relie une variable de l'objet de simulation à une ou plusieurs variables du composant graphique correspondant.

Lors de l'animation, le traducteur vérifie de manière séquentielle la stabilité de chaque contrainte. Pour chaque cas d'instabilité, il lance la méthode de traduction correspondante afin de satisfaire la contrainte.

Dans l'animation d'algorithmes, nous n'employons pas la même méthode car c'est l'algorithme qui gère l'information. L'objet de simulation ne peut être alors le modèle d'un processus (industriel ou autre) dont la partie dynamique, définie sous forme de contraintes, représente le comportement de celui-ci. En fait, l'objet de simulation joue dans notre cas simplement le rôle d'interface entre l'information que l'on veut animer et le traducteur.

L'objet de simulation est lié avec les variables du programme par l'intermédiaire d'un dictionnaire de la même manière que le modèle présenté précédemment dans la visualisation de synthèse. Quand un changement de variables est détecté, l'objet de simulation reçoit le nouvel état de ces variables et demande au traducteur de lancer et d'exécuter la ou les méthodes de traduction correspondantes. Ceci a pour conséquence de rafraîchir la présentation en lançant l'animation.

III.12 - L'EDITEUR

L'écriture des programmes s'effectue par l'intermédiaire d'une fenêtre texte présentée en figure 3.21 avec toutes les interactions de base d'un éditeur, copier, coller, sauvegarde des programmes sous forme de fichiers, etc.

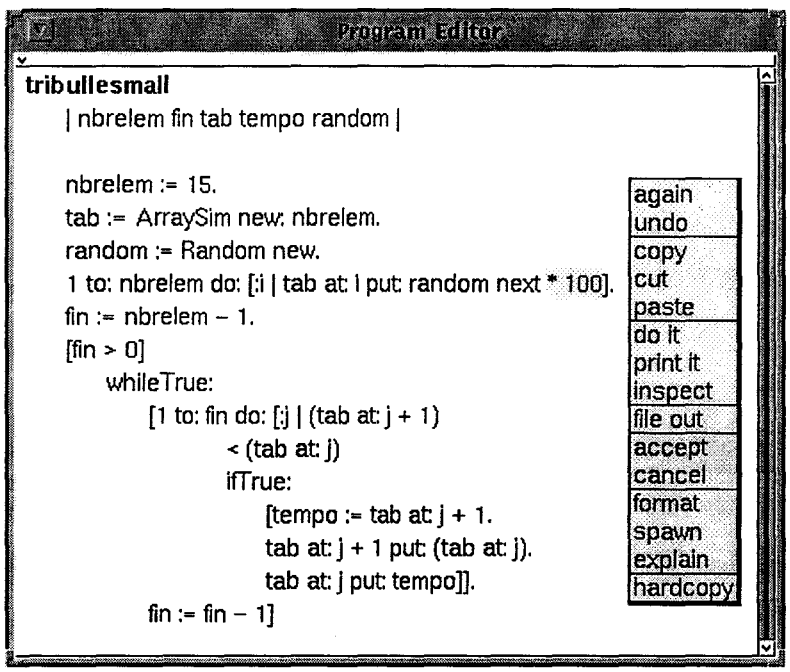


Figure 3.21 : Présentation de l'éditeur et du menu fugitif associé.

Cet éditeur permet d'écrire et de compiler de manière transparente pour l'utilisateur tout algorithme écrit en employant la syntaxe du langage Pascal ou Smalltalk.

III.13 - LE MENU GENERAL

Un des grands soucis que nous avons eu tout au long du développement de notre simulation a été de conserver un système homogène en ce qui concerne l'interface homme-machine. Ainsi, une fenêtre autonome fournit suivant le choix de l'utilisateur, un menu horizontal ou vertical qui lui permet d'accéder aux différents outils que nous avons développés (Cf. figure 3.22).

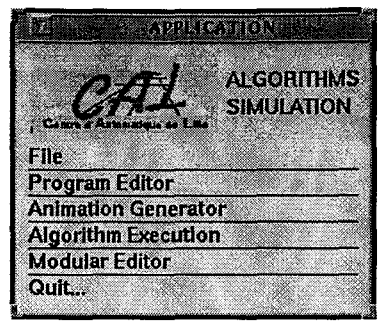


Figure 3.22 : *Présentation verticale du menu général.*

Ce menu permet à l'utilisateur d'accéder ou de créer des fichiers programmes, de les éditer et de les exécuter. Il permet également d'accéder aux outils générant les animations graphiques qui seront attachées aux données visualisées ou aux algorithmes, d'accéder à la plate-forme de programmation modulaire (présentée dans le chapitre IV) et enfin de quitter le logiciel.

III.14 - STRUCTURE ET INTERACTIONS DU SYSTEME PROPOSE

La structure du système de visualisation de programmes proposé est présentée en figure 3.23 faisant apparaître les différents liens qu'ont les outils entre eux et leurs natures.

Par l'intermédiaire de l'éditeur, l'utilisateur peut enrichir et modifier la bibliothèque d'algorithmes puis traiter un de ceux-ci par l'entremise de la fenêtre de simulation ou d'exécution d'algorithmes. Lors de cette exécution, l'utilisateur peut accéder aux différentes visualisations d'information qui lui ont traits dans des fenêtres autonomes.

Pour connaître leur représentation et animation graphique, la visualisation d'une donnée ou d'une expression fait automatiquement appel à la classe de l'objet à afficher ou pour la visualisation de synthèse à l'algorithme. Ces animations sont constituées par l'intermédiaire de la fenêtre d'édition graphique et de prototypage puis associées aux algorithmes ou aux classes du système par l'intermédiaire de l'objet de simulation du modèle P.Os.T.

Un exemple de séquence d'utilisation des outils par un utilisateur expérimenté a été ajouté à la figure 3.23 (signalée par une séquence de chiffre).

Après avoir créé un programme par l'intermédiaire de l'éditeur, l'utilisateur constitue les structures graphiques de sa visualisation de synthèse et des données non établies puis leur associe des animations. Ceci lui permet d'exécuter le programme et de visualiser les différentes informations demandées.

La dernière action concerne les interactions que l'utilisateur peut effectuer avec la simulation. Celles-ci sont multiples et les plus importantes (en dehors des différents modes de simulation, des facilités de présentation, etc.) sont les suivantes :

- Agir sur une structure de données du programme, c'est-à-dire changer la valeur des données de l'algorithme par l'intermédiaire des deux inspecteurs de la simulation. Il est certain que cette possibilité d'interaction directe en cours d'exécution peut conduire à des situations aberrantes que l'algorithme ne pourra jamais rencontrer. Ce changement de valeur entraîne automatiquement une remise à jour de l'image graphique de cette donnée, si celle-ci est visualisée.

- Changer le code source d'une méthode ou d'une procédure. Cette interaction a été facilement permise car Smalltalk est un langage semi-interprété et compile ces messages indépendamment les uns des autres. Ainsi, si nous changeons le texte d'une méthode en cours d'exécution dans le débogueur, nous ne recompilons que celle-ci sans remettre en cause les traitements antérieurs. Les pointeurs des messages précédents empilés n'ont pas été réaffectés.

- Visualiser, dans des fenêtres autonomes d'animation de données, la valeur d'une expression sélectionnée à la souris dans le texte source ou définie textuellement par l'utilisateur.

- Visualiser différents contextes ce qui permet à l'utilisateur de regarder l'état courant d'exécution de chacune des méthodes empilées et également de reprendre l'exécution du programme en sélectionnant un contexte antérieur à celui présenté.

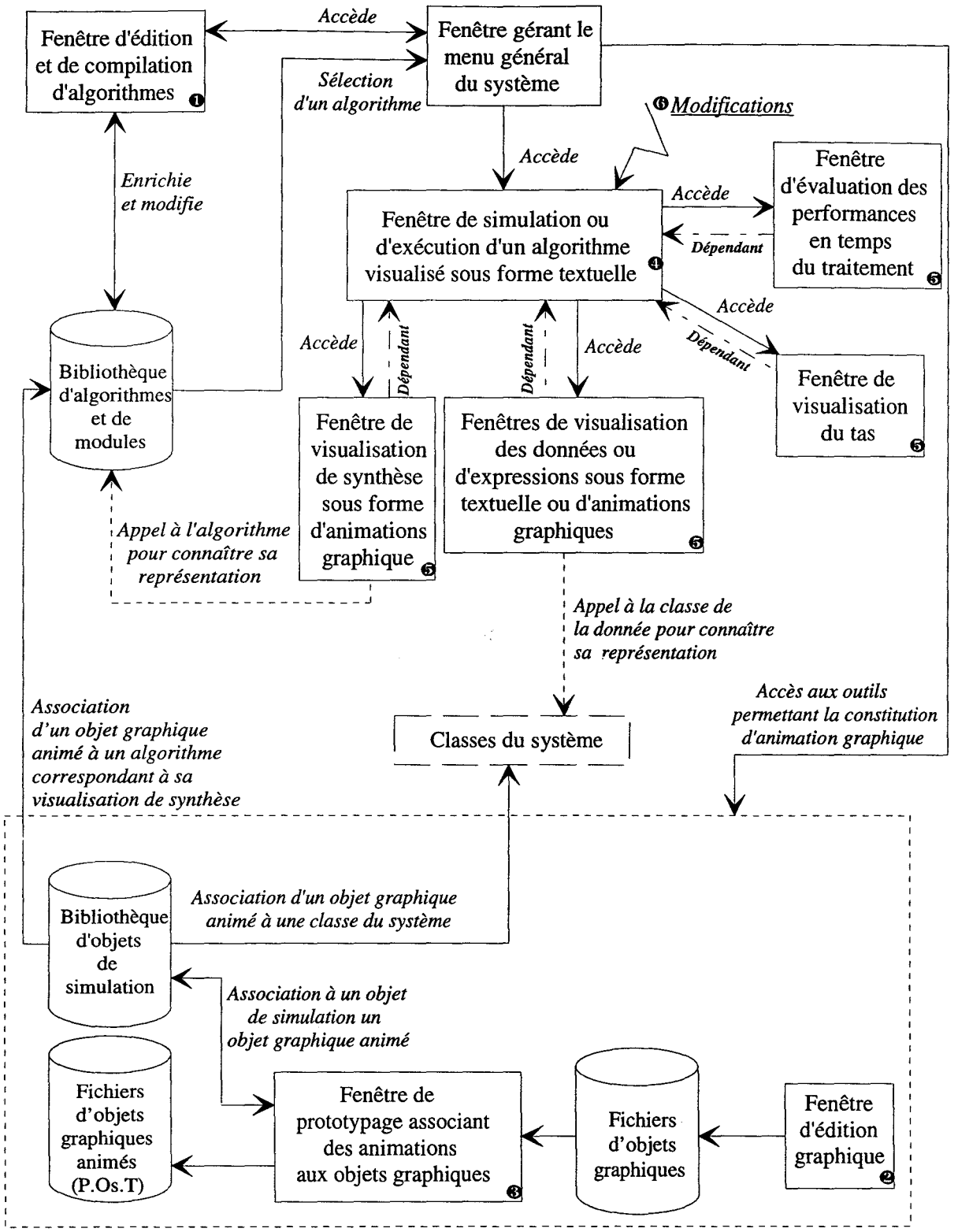


Figure 3.23 : *Structure du système de visualisation d'algorithmes proposé*

III.15 - CONCLUSION

L'originalité de notre système est l'environnement convivial que nous avons développé pour écrire et animer les algorithmes.

Cette convivialité est principalement caractérisée par la présentation des informations d'un programme dans des fenêtres autonomes, ce qui permet à l'utilisateur d'organiser son propre scénario. Celui-ci peut écrire textuellement, dans un langage donné (Smalltalk ou Pascal), n'importe quel algorithme et facilement visualiser les différentes informations qu'il comporte selon qu'il les juge nécessaires à sa compréhension.

Plus explicitement, l'utilisateur peut visualiser, sous sa forme textuelle, l'exécution de l'algorithme avec l'affichage des expressions en cours d'évaluation ; visualiser l'état des différentes structures de données ainsi que de n'importe quelles expressions du programme et obtenir une visualisation de synthèse représentant le comportement de l'algorithme de manière graphique et animée.

D'autre part, la remise à jour de la représentation de ces informations s'effectue en faisant intervenir des relations de dépendance qui permettent de gérer les événements sans altérer le texte originel du programme. Cette technique contribue à obtenir une simulation interactive car l'animation ne dépend pas directement du code source du programme.

La construction de ces animations se fait par l'intermédiaire d'une interface conviviale et interactive basée sur le modèle d'architecture P.Os.T. (Présentation, Objet de Simulation, Traducteur) et donc sans connaître au préalable de bibliothèque d'animation ou de langage graphique spécialisé.

Enfin, l'utilisateur peut agir de façon simple et conviviale, à n'importe quel instant, sur les données du programme, sur le code source, ..., et conserver une cohérence entre l'image visualisée et l'état du programme.

Le système interactif proposé permet alors à l'utilisateur de visualiser les effets induits de ces changements, pour comprendre par exemple comment chacune des données de l'algorithme affecte son comportement.

Cependant, si les différents outils et concepts présentés dans ce chapitre nous permettent de visualiser toutes les informations d'un algorithme quelconque, ceci ne peut être établi facilement que pour des programmes restreints. Pour réduire la complexité de représentation d'un programme, l'idée est de le décomposer en différents modules permettant de visualiser, indépendamment les uns des autres, les informations qu'ils comportent.

Mais pour qu'un programme soit visualisable par parties, il est nécessaire qu'il soit composé en employant les mêmes concepts de décompositions structurelles. D'autre part, qui dit programmation modulaire dit réutilisabilité de ces composants.

Pour obtenir ces différentes facilités de visualisation et de composition en conservant un environnement convivial et interactif, nous avons développé un outil de programmation graphique spécialisé présenté au prochain chapitre constituant une plate-forme de programmation visuelle autonome.

CHAPITRE IV

L'OUTIL DE PROGRAMMATION VISUELLE DÉVELOPPÉ ET LA VISUALISATION DE PROGRAMMES

Pour résoudre les problèmes de taille conséquente et atteindre des objectifs d'extensibilité, de réutilisabilité et de compatibilité, la notion de composants logiciels ou modules interconnectés selon un protocole établi est introduite et est obtenue par l'intermédiaire d'une plate-forme de programmation visuelle.

IV.1 - LES OBJECTIFS ET LES ASPECTS MODULAIRES D'UN PROGRAMME

La programmation modulaire a été parfois définie comme la construction de programmes par assemblage d'éléments de petite taille, en général des sous-programmes. Du reste, une des méthodes de conception qui attache de l'importance aux structures modulaires est la "conception structurée" [YOU 79].

Le succès de la modularité vient du fait que la décomposition d'un programme en composants individuels permet de réduire sa complexité [PAR 72]. Mais la modularité implique nécessairement la réutilisabilité des composants logiciels que nous créons. Le principe étant que les programmeurs ne passent plus leur temps à récrire certains motifs de base : tri, recherche, lecture, écriture, comparaison, parcours, etc.

Des efforts dans ce sens ont déjà été réalisés aux Etats-Unis par le programme "STARS" [NSI 85] ("Software Technology for Adaptable, Reliable Systems", technologie logicielle pour des systèmes adaptables et sans erreurs) dont l'un des buts est de fournir une bibliothèque de composants Ada réutilisables.

L'intérêt pour nous de considérer une telle programmation à deux objectifs principaux :

- Développer de nouveaux programmes en associant différents algorithmes élémentaires ou modules entre eux, issus d'autres programmes ou conçus spécifiquement pour le résoudre. Cette forme de programmation peut aider les chercheurs et les concepteurs de programmes à explorer plus facilement et plus rapidement de nouvelles variantes d'algorithmes.

- Faciliter la compréhension de l'utilisateur en décomposant la problématique d'un algorithme en modules ce qui permet de ne visualiser que le comportement et les caractéristiques d'un sous-problème [BAL 96]. De ce fait, le programmeur peut agir plus facilement sur une petite partie d'un programme en occultant certaines opérations.

Pour conserver un système homogène, la réalisation de ces deux objectifs doit se faire dans un environnement interactif et convivial.

L'interactivité du système doit être la plus large possible en nous attachant plus particulièrement à développer les concepts nous permettant l'interaction d'altération.

Nous devons, par exemple permettre à l'utilisateur de changer la valeur de chacune des données ainsi que le code des différents modules employés indépendamment les uns des autres.

De même, la convivialité du système doit être sauvegardée par l'emploi d'une interface visuelle et des concepts qui lui ont trait. Celle-ci facilite alors le processus de programmation en ne faisant appel à aucune initiation ou connaissance particulière et peut être, par conséquent, adaptée à un utilisateur naïf. C'est pourquoi la conception de programmes à partir de modules élémentaires doit employer des règles syntaxiques visuelles simples, représentatives et facilement compréhensibles. Ces programmes seront concrètement implémentés en assemblant de manière graphique et abstraite différents composants logiciels constituant une forme de programmation visuelle [CHA 90 ; GLI 90a ; GLI 90b].

Mais avant de concevoir une plate-forme de programmation modulaire, il nous faut déterminer une base conceptuelle pour établir les formes des modules employés et les différents critères qui les caractérisent.

IV.2 - FORMES ET CRITERES DE MODULARITE

Comme nous l'avons introduit (Cf. chapitre I.1.3), un algorithme qui peut être intégré à un autre est un module. De ce fait, un module est un algorithme dit élémentaire qui peut être conçu indépendamment du contexte dans lequel il doit être utilisé.

Historiquement, la notion de module a été introduite pour résoudre les problèmes d'analyse et d'implantation de grands logiciels définissant de larges et complexes programmes. Ceux-ci correspondent pour notre application à au moins plus d'un seul traitement différent par algorithme conçu.

On a fréquemment remarqué que la complexité de ces programmes prenait la forme d'une hiérarchie, dans laquelle un système complexe est composé de sous-systèmes interconnectés possédant eux-mêmes leurs propres sous-systèmes et ainsi de suite jusqu'à ce que le niveau du composant élémentaire soit atteint [COU 85a]. Mais le choix désignant un composant d'un système comme étant élémentaire est totalement arbitraire et dépend complètement de l'observateur du système.

D'autre part, si le fait de décomposer un logiciel en composants individuels permet de réduire sa complexité, la modularité est aussi un élément clé de la réutilisabilité et de l'extensibilité.

La décomposition en modules doit alors être prise en compte lors de l'analyse puis lors de la conception pour se répercuter enfin sur la programmation.

IV.2.1 - Analyse et conception modulaire

La synthèse des divers conseils recensés par Booch [BOO 91] préconise la construction de modules aussi indépendants les uns des autres que possible. Chaque module doit regrouper le maximum de données liées au même type abstrait. Booch propose la définition suivante :

"La modularité est la propriété d'un système qui est décomposé en un ensemble de modules cohérents et faiblement couplés" [BOO 91].

Meyer [MEY 88] propose cinq critères pour évaluer le degré de modularité des méthodes de conception. Ces critères sont la décomposabilité, la composabilité, la compréhensibilité, la continuité et la protection.

1) La décomposabilité modulaire est un critère très exigeant qui n'est réellement satisfaisant que lorsque les sous problèmes obtenus après une conception descendante peuvent réellement être traités par des personnes travaillant séparément.

2) Le critère de composabilité modulaire doit conduire à l'obtention de modules qui puissent être combinés les uns avec les autres, afin de résoudre de nouveaux problèmes. Ce critère est directement issu du concept des composants logiciels proposés et souhaités par Cox [COX 86] pour aboutir à la réutilisabilité du logiciel. Le but de la composabilité est bien de pouvoir assembler les différents modules conçus pour obtenir de nouveaux composants pouvant eux aussi être combinés. Un des exemples de composabilité proposé par Meyer [MYE 88] est celui du langage de commande d'Unix, où l'utilisation des tubes permet de composer les commandes, chaque processus pouvant utiliser sa sortie standard pour communiquer avec l'entrée standard du processus qui suit.

3) La compréhensibilité modulaire est importante en particulier pour la maintenance. Chaque module doit pouvoir être compris séparément par une personne qui ne fait pas forcément partie de l'équipe de développement initial. Un exemple de ce qu'il faut éviter est donné par Meyer, qui conseille de ne pas créer de systèmes à dépendances séquentielles dans lesquels les modules doivent s'enchaîner dans un ordre précis.

4) La continuité modulaire est respectée si une modification minime de spécification n'entraîne la modification que d'un seul, voire de peu de modules. Ce critère a trait au couplage des modules. Un exemple est donné par Meyer en ce qui concerne les constantes symboliques. Si lors d'un développement toutes les constantes numériques ou alphanumériques ne sont utilisées qu'au travers de noms symboliques, la modification d'une valeur n'entraînera que la modification de sa définition. Il prend également comme exemple le principe de la référence uniforme [GES 75] qui consiste à pouvoir accéder à une donnée, que ce soit par stockage ou par calcul, en ayant la même notation d'appel.

5) La protection modulaire concerne la gestion des exceptions et autres conditions de fonctionnement dégradé. La protection modulaire est respectée si une exception survenant à l'exécution dans un module ne se propage pas à d'autres modules. Dans le pire des cas il est acceptable qu'elle se propage à quelques autres modules.

Les critères présentés par Meyer le conduisent à énoncer cinq principes :

1) Des unités linguistiques modulaires impliquent que "Les modules doivent correspondre à des unités syntaxiques du langage." Ce langage peut être un langage de programmation, de conception, de spécification, etc., et dans ce cas, les modules doivent être compilables séparément [FEL 79].

2) Peu d'interfaces entraîne que "Tout module doit communiquer avec aussi peu d'autres que possible."

3) Peu d'interfaces (couplage faible) implique que "Si deux modules communiquent, ils doivent échanger aussi peu d'informations que possible."

4) Des interfaces explicites entraînent que "Chaque fois que deux modules A et B communiquent, cela doit ressortir clairement du texte de A, de B ou des deux."

5) Le masquage de l'information entraîne que "Toute information concernant un module doit être privée sauf si elle est explicitement déclarée publique." Ce principe a été introduit dans un article de référence par Parnas [PAR 72].

Meyer complète son tour d'horizon sur les modules, par la notion d'ouverture et de fermeture d'un module. Un module est dit fermé dès qu'il peut être utilisé par des modules clients. Un module est dit ouvert lorsqu'il peut encore être étendu. Il constate d'autre part que seule l'approche orientée objet, avec les techniques d'héritage, autorise une ouverture et une fermeture simplifiées des modules en cours d'élaboration, et ce, grâce aux faibles dépendances entre les différents objets.

Mais, dans ce cas, le type d'interface obtenu entre les différents modules est avant tout organisé suivant une structure d'analyse programmatique, ce qui rend plus complexe son utilisation pour un utilisateur naïf. Il nous faut donc établir une méthodologie d'analyse et de conception indépendante des langages de programmation utilisés pour obtenir un modèle de programmation modulaire satisfaisant à nos contraintes.

En dehors des méthodes qui s'appliquent à une catégorie de langage comme les méthodes objets, il existe trois grandes catégories de méthodes d'analyse et de conception [COA 91] :

- Les méthodes de décomposition fonctionnelle sont caractérisées par une stratégie d'analyse et de conception qui se concentre sur les traitements que doit réaliser le système à concevoir. Ces traitements sont décomposés en étapes et sous étapes et l'application d'un traitement spécifique est soumise à un protocole (Cf. figure 4.1).

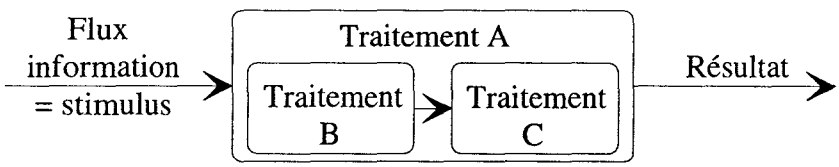


Figure 4.1 : Représentation d'une analyse par décomposition fonctionnelle

- Les méthodes basées sur une approche par flot de données sont souvent appelées "techniques d'analyse structurée" [YOU 89] dont la méthode SADT (Structured Analysis and Design Technique") est un exemple particulièrement adapté aux contraintes de l'informatique technique [IGL 89]. Une représentation de cette méthode d'analyse est présentée en figure 4.2.

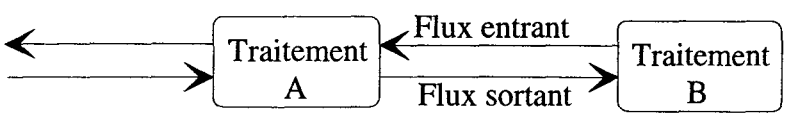


Figure 4.2 : Représentation d'une analyse par flot de données

- Les méthodes basées sur la modélisation de l'information dont l'approche a considérablement évoluée depuis l'apparition des diagrammes E/R (Entité / Relation) de Chen [CHE 76] (Cf. figure 4.3) jusqu'aux modèles de données sémantiques [MEL 88]. La méthode systémique Merise [ROC 89] accorde par exemple une large part a la modélisation des données selon un schéma inspiré de celui de Chen.

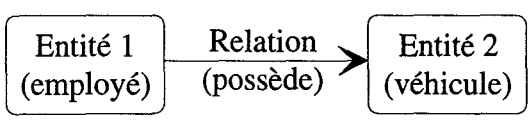


Figure 4.3 : Un exemple d'une représentation d'une analyse de l'information

Ces trois catégories de méthodes ont leurs avantages et leurs inconvénients.

Le modèle d'analyse de décomposition fonctionnelle permet d'identifier les différents traitements que doit réaliser un programme en répondant aux critères définis par Meyer mais les relations entre ces traitements définissent des événements correspondant à des flux physiques de matière, d'argent, de personnel, etc., ce qui ne correspond pas à l'information que l'on veut véhiculer.

Le modèle de flot de données permet de rendre explicite la communication entre deux traitements en montrant les dépendances entre les données en entrée et celles en sortie des processus chargés de réaliser les fonctions précises de L'algorithme. L'inconvénient de ce modèle est que deux traitements peuvent s'échanger mutuellement des informations sans avoir établi d'ordre d'évaluation.

La modélisation de l'information ne représente que des relations statiques entre deux entités et ne peut correspondre à une communication dynamique entre des processus de traitements.

En combinant le modèle de décomposition fonctionnelle et le modèle de flot de données en interdisant les échanges mutuels d'informations nous obtenons un modèle d'analyse et de conception permettant de répondre aux critères et aux principes 1, 2, 4 et 5 énoncés par Meyer dont un exemple de représentation est en figure 4.4.

Ce modèle graphique prend en compte les aspects fonction de transformation d'informations des logiciels et s'inspire de la notation des schémas E/R de Chen pour les diagrammes à flots de données.

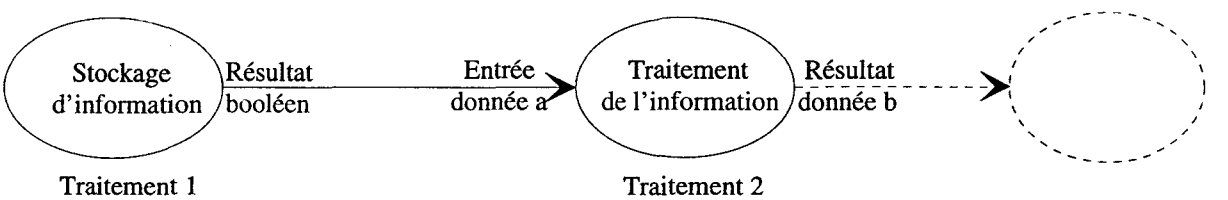


Figure 4.4 : *Représentation de notre modèle d'analyse et de conception modulaire pour un programme particulier*

Par contre, le modèle a tendance à multiplier le nombre de communications inter-modules qui peut gêner leur réutilisabilité.

IV.2.2 - Problèmes de communications inter-modulaires

On peut constater que les communications possibles entre modules selon leur nature, peuvent être de types divers. Les modules peuvent s'appeler les uns les autres (si ce sont des procédures) et peuvent partager des structures de données, etc.

Mais, quelle que soit la nature des modules choisie, le principe du peu d'interface ($3^{ème}$ critère de Meyer) limite le nombre de ces connexions.

Plus précisément, si un système est composé de *n* modules, le nombre de connexions inter-modules doit être beaucoup plus près du minimum, c'est-à-dire *n-1*, que du maximum, *n (n-1)/2*.

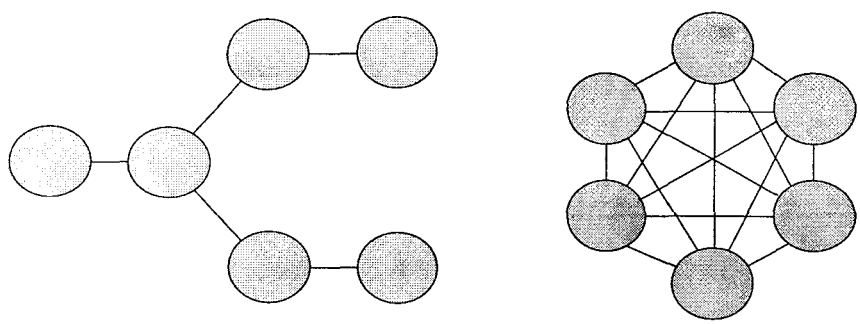


***Figure 4.5** : Types de structures d'interconnexion de modules,(a) nombre de connexions minimum (5 pou 6 modules), (b) nombre de connexions maximum (15 pour 6 modules).*

Ce principe découle des critères de continuité et de protection : il faut un minimum de relation pour ne pas propager une erreur. Il est aussi lié à la composabilité car il ne doit pas dépendre d'un trop grand nombre d'autres modules pour être de nouveau utilisable.

L'objectif de composabilité, dont le critère a été énoncé précédemment (Cf. paragraphe IV.2.1), est en relation directe avec le problème de la réutilisabilité : il s'agit de trouver les moyens de concevoir des éléments logiciels qui réalisent des tâches définies et qui soient utilisables dans des contextes très différents.

Toutefois, l'avantage du modèle fonctionnel est qu'il est basé, d'une part sur une décomposition naturelle d'un problème permettant de distinguer facilement dans un programme les modules qui le constituent, et d'autre part sur des concepts graphiques qui facilitent son implémentation dans un environnement visuel par transposition directe.

L'utilisation d'une plate-forme de programmation visuelle permet à l'utilisateur de construire des programmes basés sur le modèle décrit ci-avant et également de bénéficier des concepts d'interactivité et de convivialité liés à ce type d'interface.

La conception de programmes employant un modèle graphique de composition modulaire permet alors d'aboutir à un vieux rêve : transformer le processus de conception du logiciel en un jeu de construction où les programmes peuvent être élaborés à partir d'éléments standards et reliés simplement [VOS 86 ; GOL 91 ; YOU 95a].

Le modèle d'analyse et de conception présenté en figure 4.4 est un modèle graphique basé sur les flux de données fait naturellement appel au paradigme de programmation visuelle employant cette technique.

IV.3 - LA PROGRAMMATION VISUELLE BASEE SUR LE MODELE DE FLUX DE DONNEES

Dans les langages de programmation visuelle basée sur les flux de données les programmes sont représentés par un graphique direct où les noeuds représentent des fonctions et où les arcs représentent le flux de données entre les fonctions. Les arcs qui vont à un noeud représentent les données d'entrées d'une fonction ; les arcs qui sortent représentent les données de sorties. Nous pouvons citer comme langages de programmation visuelle employant le modèle de flux de données Viz [HER 96], PROGRAPH [GOL 91], Show and tell [KIM 86], etc.

Ce modèle a été employé par de très nombreux systèmes qui ont révélé les avantages et les inconvénients de celui-ci.

IV.3.1 - Avantages du modèle

Le modèle de flux de données à pour avantage d'être :

- performant pour des domaines d'application spécifique (par exemple pour la collecte de données et l'analyse, LabView [VOS 86]) ;
- très bien adapté pour des novices et des non-programmeurs car il permet de composer des programmes avec un apprentissage restreint ;
- adapté pour la manipulation de données (traitement d'images, traitements graphiques, visualisation scientifique) ;

- approprié pour la transformation des données (filtrage, CANTATA [YOU 95a, YOU 95b]);
- efficace s'il est basé sur une vaste bibliothèque de fonctions tout en permettant d'en construire de nouvelles.

IV.3.2 - Inconvénients du modèle

Si le modèle de flux de données présente de nombreux avantages il comporte également des inconvénients qui sont :

- problème de gestion de l'espace sur l'écran car généralement le nombre d'éléments visuels définis est limité à un nombre fini pour permettre de les afficher sans les superposer ;

- généralement pas suffisamment clair quand la construction de programmes fait appel à de très nombreuses fonctions ou quand celles-ci doivent être fournies par l'utilisateur ;

- pas assez performant pour des programmeurs professionnels qui ne peuvent élaborer des algorithmes faisant appel à des structures spécifiques ou ayant de très nombreuses communications entre les fonctions.

IV.3.3 - Extension du modèle

Pour répondre aux différents inconvénients du modèle de base, celui-ci peut être étendu de la manière suivante :

- construction de fonction itérative spécifique du type cyclique (for, while, ...), conditionnel (if, ...) , parallèle ;

- construction d'abstraction procédurale où un sous-graphe est condensé en un simple noeud permettant une meilleure visualisation d'importants programmes ;

- construction de séquencement (utilisé par LabView), vérification de typage sur les arcs, etc.

IV.3.4 - Modes d'exécution associés

Pour caractériser les différents modes d'exécution d'un modèle de flux de données Tanimoto [TAN 90] propose une distinction entre trois niveaux :

- significative (exécution à la demande, Pict [GLI 84]),
- indifférente (lors d'une ré-exécution automatique, HI-Visual [HIR 91]),
- et vivante (exécution continue, VIVA [HIL 92]).

Mais principalement les modèles de flux de données peuvent être vus comme s'exécutant de deux manières différentes : dans le premier cas, les noeuds sont évalués dès que toutes les entrées deviennent disponibles ; dans le second cas ils sont évalués suivant la demande d'une requête explicite faite sur un noeud spécifique. Le système CANTATA offre ces deux alternatives.

IV.3.5 - Conclusion

La détermination des avantages et inconvénients du modèle de flux de données nous a permis de déterminer les éléments fonctionnels indispensables pour élaborer notre propre interface. Cependant, la plupart des systèmes tels que : ARK [SMI 87], Fabrik [ING 88], VIPR [CIT 95], etc., sont basés sur la définition d'un nouveau langage de programmation visuelle évolué ce qui amoindrit leur portée pour des utilisateurs naïfs.

Nous nous sommes ainsi efforcés de pallier cet inconvénient en employant des concepts de programmation naturellement visuelle et facilement compréhensible.

IV.4 - L'OUTIL DE PROGRAMMATION VISUELLE DEVELOPPEE [VAN 96B]

Le modèle fonctionnel, comme nous venons de le voir, repose sur une décomposition d'un problème en procédures de traitement pouvant se rapporter à des modules ou des composants logiciels dont les dépendances entre les données sont représentées sous forme de flots de données. Ce modèle visuel nous permet alors de concevoir et visualiser par partie de grands programmes dans un environnement de programmation spécialisé.

L'interface de programmation visuelle que nous avons conçue permet à un utilisateur de créer un programme à partir d'algorithmes élémentaires sous forme graphique. Les originalités de cet outil sont multiples mais la principale est l'incorporation à l'environnement des outils de visualisation de programmes et des caractéristiques qui leur sont intrinsèques. Ainsi, après avoir établi un programme de manière visuelle, l'utilisateur peut visualiser le traitement d'un ou plusieurs modules employés, visualiser l'état de ces données sous forme graphique et animée, modifier au cours de l'exécution ces différentes informations, etc.

La structure de l'interface de programmation et de visualisation présentée en figure 4.6 met en évidence les interactions que la fenêtre de programmation visuelle peut avoir avec les outils de visualisation de programme et les liaisons avec ses structures.

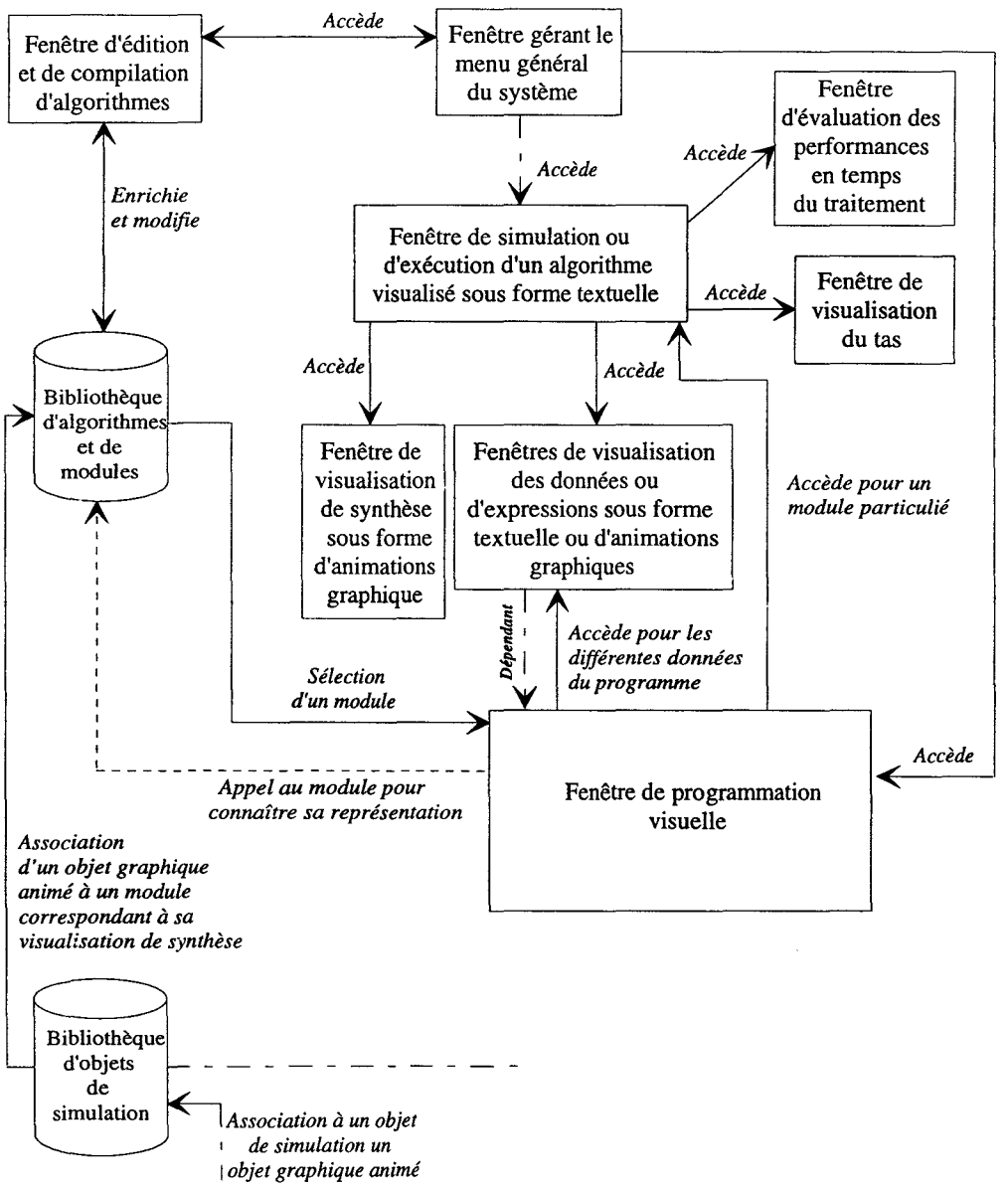


***Figure 4.6** : Structure de l'interface de programmation et de visualisation*

Tout au long du développement de cet outil, nous avons ainsi eu le souci de développer les concepts nécessaires pour combiner, aussi bien au niveau conceptuel que pragmatique, les outils de visualisation de programmes et de programmation visuelle pour créer un espace visuel de travail homogène.

Nous allons dans la suite de ce chapitre expliciter ces concepts et définir la gestion et la représentation des modules et leurs liaisons avec le système de visualisation de programme présenté au chapitre III.

IV.4.1 - Choix des modules

La réutilisabilité des éléments implique qu'un module conçu doit pouvoir être intégré à tout algorithme qui en a besoin. Il est de ce fait utile de construire une bibliothèque de modules, telle que des modules de tri, de résolution d'équations, de traitement de matrices, etc.

La bibliothèque de modules dans notre environnement a la particularité d'être constituée, a priori, de tous les algorithmes écrits et animés avec l'outil de visualisation de programme et de pouvoir être également enrichie par l'implémentation de nouveaux modules en écrivant des algorithmes élémentaires par l'entremise de l'éditeur présenté au chapitre III.12.

Les modules sont choisis par l'intermédiaire d'une boîte de dialogue particulière présentant la bibliothèque sous forme de listes et de sous-listes correspondant aux objets et messages de l'environnement ou par analogie à une programmation structurée aux programmes et sous-programmes (Cf. figure 4.7).

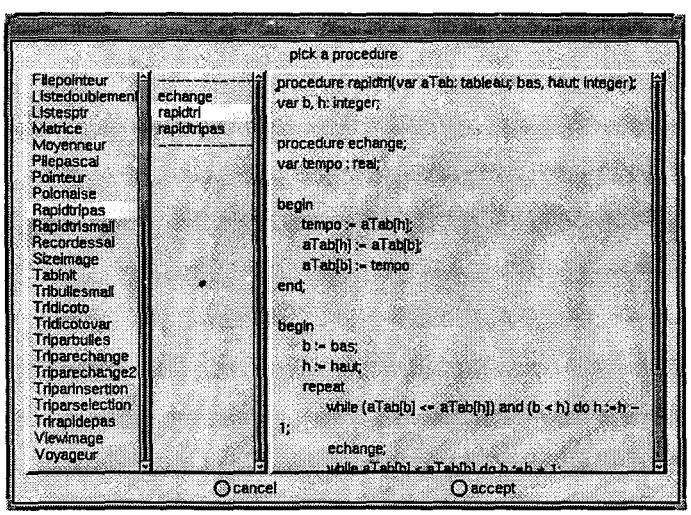


***Figure 4.7** : La fenêtre de dialogue permettant de sélectionner un module autonome.*

La fenêtre de dialogue est en fait décomposée en trois sous-vues. La première présente une liste de tous les objets ou "programmes" qui peuvent être exécutés et visualisés indépendamment de tous les autres algorithmes. La sélection d'un élément de cette liste présente dans la seconde sous-vue les messages ou les "sous-programmes" constituant l'algorithme élémentaire. La sélection d'un de ces éléments affiche dans la troisième sous-vue le code du message.

La présentation de ce code permet à l'utilisateur de visualiser l'algorithme qui sera effectivement exécuté si celui-ci accepte le module ainsi sélectionné. Si ce choix est validé, il y a automatiquement fermeture de la fenêtre et apparition de ce module dans l'outil de programmation visuelle sous la forme d'une représentation particulière.

IV.4.2 - Représentation d'un module

La visualisation du code des modules employés, dans la simulation modulaire, peut être dans un premier temps considérée comme secondaire car lors de la construction d'un nouveau programme, nous nous intéressons plus particulièrement à ce qu'effectue globalement l'algorithme et non comment l'algorithme est effectivement implémenté. Ainsi, chacun de ces modules peut être représenté sous une forme graphique quelconque dont la représentation la plus simple est un rectangle que l'on peut comparer à une boîte noire rendant le code non visualisable directement.

Cependant, pour qu'un utilisateur comprenne le nouveau programme écrit visuellement, il est souhaitable que l'image de ces différents modules dans l'environnement soit représentative du problème traité. Nous devons pouvoir associer à chaque module une représentation particulière correspondant à l'image de leurs comportements sous forme graphique et animée.

Il est donc logique que l'image du module dans l'outil de programmation visuelle fasse appel au même modèle d'architecture graphique orienté objet particulier baptisé P.Os.T. (Présentation, Objet de simulation, Traducteur) introduit dans l'animation d'algorithmes (cf. chapitre III.7).

L'emploi de ce modèle nous permet de bénéficier des différents atouts qui lui sont intrinsèques et dont la caractéristique principale est de pouvoir créer automatiquement, par l'intermédiaire d'une interface interactive et conviviale, une représentation et une animation sans connaître de langage graphique ou d'animation spécifique.

L'utilisateur peut ainsi, par l'intermédiaire d'une riche bibliothèque de représentations graphiques, adjoindre aux différents modules de l'environnement une image fixe ou animée, ou si celles-ci ne sont pas représentatives en créer très facilement de nouvelles.

Des exemples de représentation de modules sont présentés en figure 4.8.

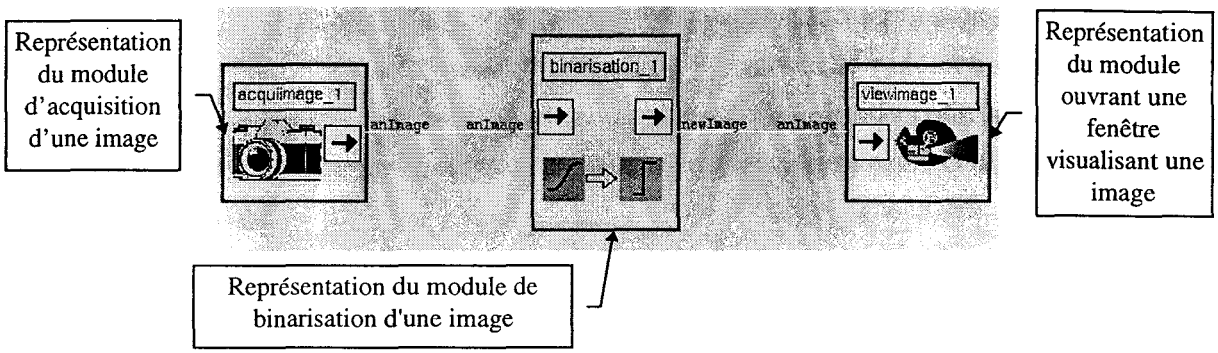


***Figure 4.8** : Présentation de différents modules employant le modèle d'animation graphique P.Os.T.*

Ayant adopté d'autre part le même modèle graphique pour visualiser les modules et les informations d'un programme, nous obtenons un environnement homogène où la représentation du comportement d'un algorithme dans la visualisation de programme peut être la même image que celle utilisée pour l'affichage du module dans la programmation visuelle.

En outre, l'objet de simulation du modèle P.Os.T. étant un objet autonome, il permet d'attacher à plusieurs modules la même représentation. Aussi, à chacun des composants logiciels de l'environnement nous associons par défaut une simple image ou, lorsque celle-ci est définie par l'utilisateur, une image originale qui s'intègre à la bibliothèque des représentations graphiques.

Si un module à la même représentation de base dans l'environnement et s'il est utilisé plusieurs fois pour constituer un programme, il est judicieux de le personnaliser informatiquement par l'intermédiaire d'un paramètre et visuellement pour savoir à quel objet l'utilisateur s'adresse effectivement au cours de la simulation. Cette personnalisation se caractérise par une identification nominative du module créée à partir de l'entête nommant le message de l'algorithme traité, et d'un numéro correspondant au nombre de composants logiciels de même nature existant dans l'environnement.

IV.4.3 - Contrôle graphique d'un module

Pour un fonctionnement cohérent de la gestion du contrôle d'interaction, un protocole de communication entre le contrôleur de la fenêtre de programmation visuelle et les modules a été établi. Ce protocole, transparent pour l'utilisateur, contient les méthodes qui permettent d'identifier les présentations ayant besoin du contrôle ou celles permettant d'attribuer le contrôle à un module quelconque du nouveau programme constitué.

Ce contrôleur est chargé d'établir la communication entre l'utilisateur et le modèle de l'environnement. Il assure ainsi les deux tâches élémentaires d'une interaction : l'affichage du programme et des informations nécessaires au dialogue d'une part, et la prise en compte des actions de l'utilisateur, d'autre part.

Pour ce faire, chacune des représentations des modules employés doit pouvoir posséder un contrôleur appelé "contrôleur graphique" que nous associons alors à l'objet "présentation" du modèle P.Os.T. (Cf. figure 4.9).

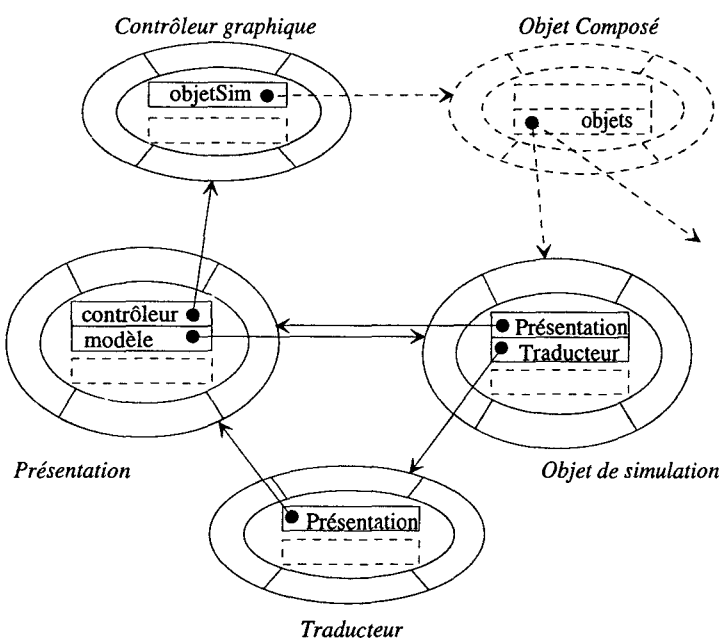


Figure 4.9 : Architecture du Modèle P.Os.T. dans la programmation visuelle.

L'accès à ce contrôleur graphique s'effectue par l'intermédiaire du dispositif de communication principale de notre application en l'occurrence la souris. Celle-ci possède donc des fonctions contextuelles différentes suivant le bouton sollicité (Cf figure 4.11).

Mais pour établir un protocole de communication cohérent entre les modules et la fenêtre de programmation visuelle, une architecture informatique de l'interface a été conçue pour faciliter ce dialogue (Cf. figure 4.10).

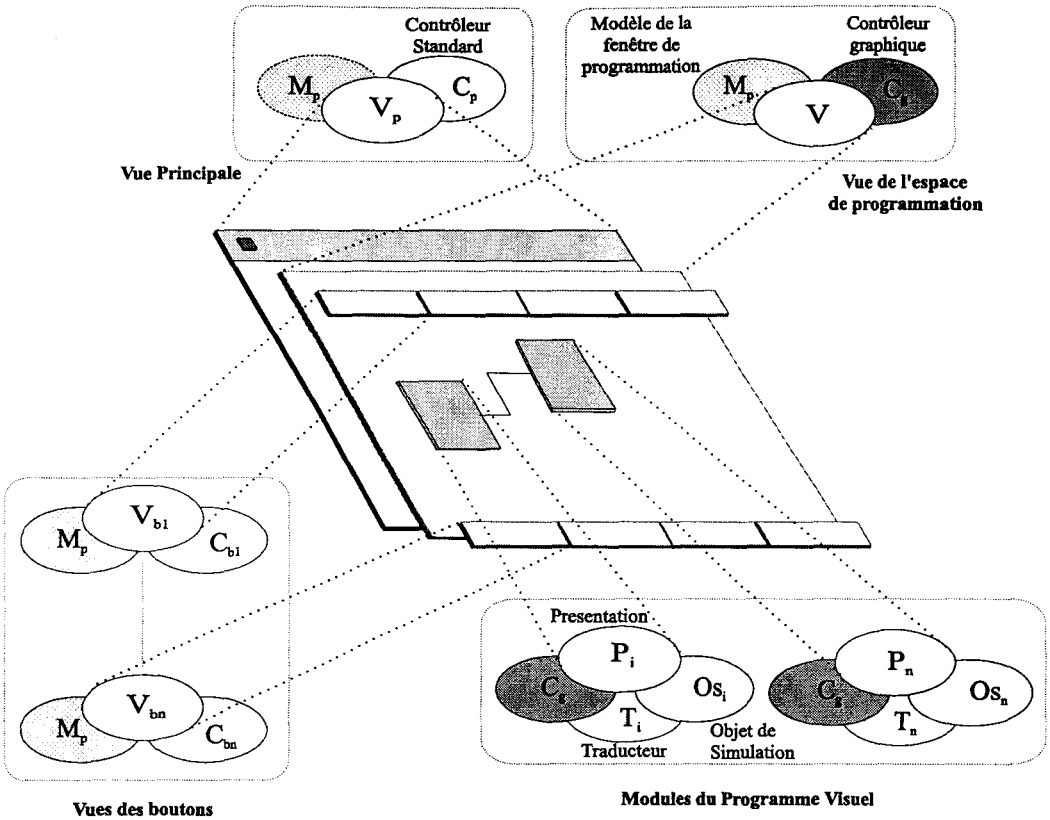


Figure 4.10 : *Architecture de l'interface de programmation visuelle.*

La vue principale de la fenêtre de programmation visuelle (ScheduledWindow) et son contrôleur (StandardSystemController) associés au modèle de la plate-forme gèrent toutes les opérations standards de contrôle concernant celles-ci (ouverture, fermeture, rafraîchissement, etc.).

Un ensemble de modèles MVC correspondant aux différents boutons gérant les fonctions de la simulation est chargé de communiquer au modèle de la plate-forme (Mp) la fonction sélectionnée. Ainsi, comme la vue de l'espace de la programmation visuelle a pour modèle "Mp", le contrôleur graphique peut s'informer de l'état de la simulation et par conséquent changer l'aspect de contrôle en fonction du bouton sélectionné.

Les actions de l'utilisateur sur ces différents boutons sont donc traduites sous forme d'événements envoyés au contrôleur graphique de la simulation. Celui-ci s'adapte, alors, en fonction du contexte courant (par défaut c'est l'édition), du bouton de fonction du menu principal sélectionné (lien, exécution), du bouton de souris sélectionné et de la position courante de celle-ci et renvoie le menu fugitif correspondant. La représentation partielle de cette interaction est présentée en figure 4.11.

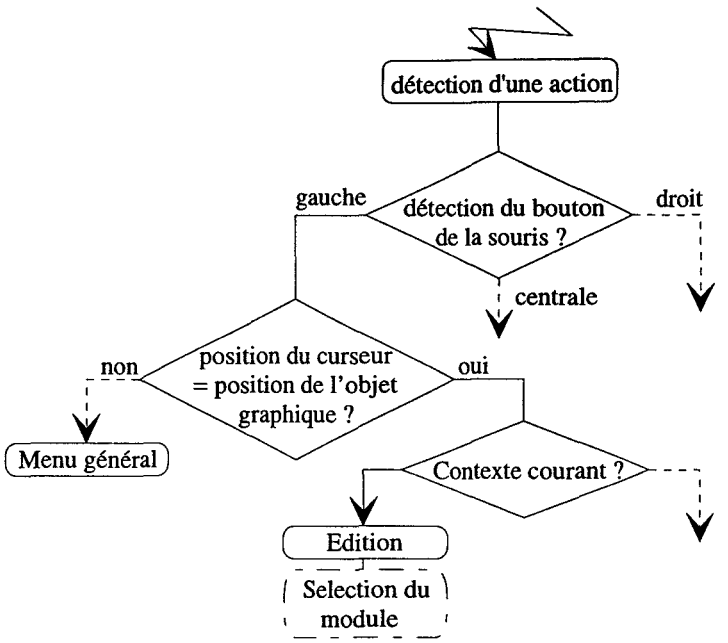


Figure 4.11 : *Organigramme de gestion des actions de l'utilisateur.*

Cette répartition de contrôle entre le contrôleur graphique de la fenêtre et les éléments de la simulation a permis l'obtention d'un dialogue par manipulation directe sur les objets affichés sur l'écran.

Ainsi, nous pouvons déplacer, détruire, dupliquer, ..., un module en le sélectionnant.

D'autre part, suivant le contexte dans lequel la programmation visuelle se trouve certaines fonctionnalités, adjointes à la fenêtre, ne peuvent être atteintes (Cf. figure 4.12).

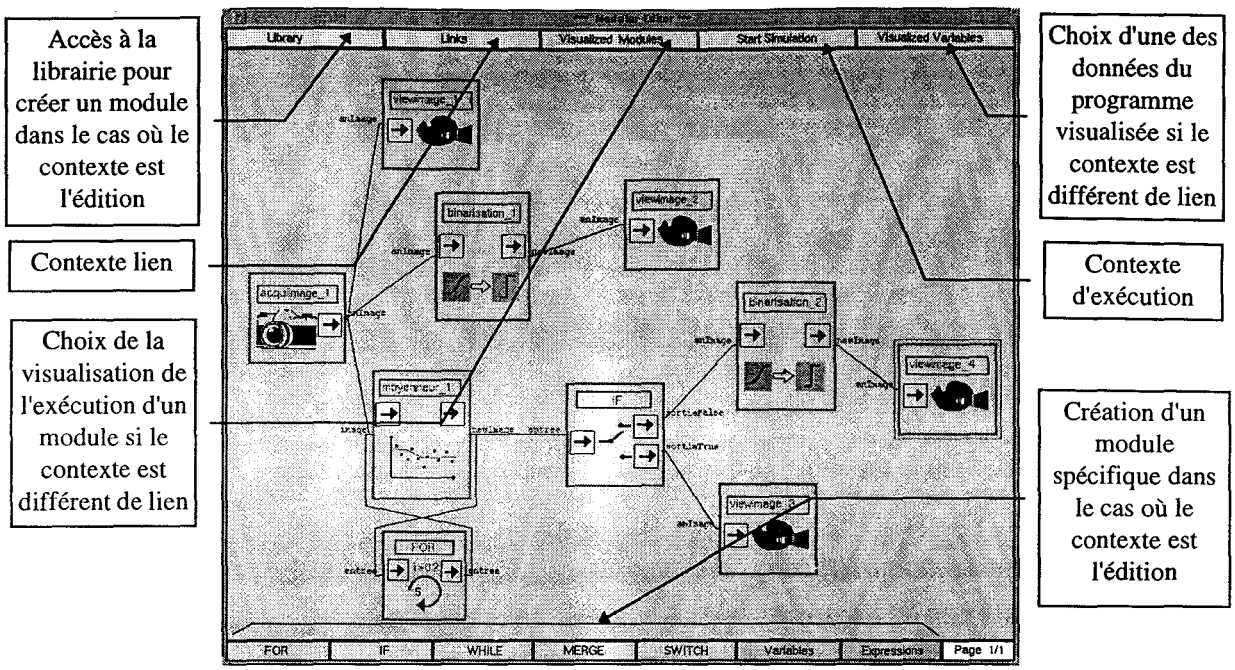


***Figure 4.12** : L'outil de programmation visuelle développé et ses contextes d'utilisation.*

IV.4.4 - Contrôle des relations entre modules

Un programme est un processus qui traite différentes données. Si nous décomposons celui-ci en différents modules, il faut introduire des relations entre ceux-ci pour qu'ils puissent transmettre les données.

Ces relations sont donc définies en déclarant les données de sortie d'un module comme les données d'entrée d'un ou plusieurs autres modules. Ces liaisons créent un réseau de relations unidirectionnelles qui contrôle le flux de données et détermine le parcours de l'exécution donnant ainsi la syntaxe de notre programme.

Les programmes créés sont des représentations directes où chaque noeud de ce graphique est un élément iconique représentant un module et chaque arc direct représente un chemin par lequel les données circulent.

Pour créer effectivement ses liaisons, le contexte "lien" est préalablement choisi par l'intermédiaire d'un des boutons associés à la fenêtre de programmation visuelle (Cf. figure 4.12). Ainsi, lors de la sélection d'un des modules que l'utilisateur désire mettre en relation, cette interaction provoque l'apparition d'un menu fugitif adapté à cette fonction, présentant en l'occurrence les variables de l'algorithme du composant. Toutefois nous ne présentons que les données concernées suivant la sélection du module d'entrée ou de sortie.

Mais avant de créer effectivement un lien, nous sommes obligés de vérifier que les données sont compatibles avec la fonction demandée (Cf. figure 4.13).

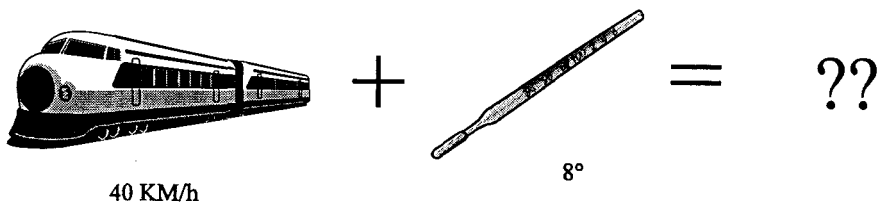


Figure 4.13 : *Contrôle de cohérence entre deux données liées.*

Ce contrôle dans le cas d'un algorithme écrit en Pascal se limite à une vérification de type entre la donnée source et la donnée destination. En Smalltalk pour qu'une donnée réponde aux différents messages que l'on pourrait lui envoyer nous vérifions que l'objet source correspond à une classe ou une sous-classe de l'objet destination.

D'autre part, la combinaison dans un même algorithme de modules écrits en Pascal et en Smalltalk, nous contraint à élaborer un protocole de communication spécifique pour les connecter. Celui-ci se caractérise principalement par une transformation de la nature des données entre les modules considérés. Par exemple, une variable de type tableau est transformée en un objet tableau.

Une fois que les données sont compatibles et cohérentes, la liaison créée se matérialise graphiquement par l'apparition d'une simple ligne entre les deux modules (Cf. figure 4.15). Celle-ci est redessinée automatiquement si l'un ou l'autre des deux objets est déplacé, ou supprimé si nous en détruisons un.

La suppression d'un lien s'effectue en fait de deux façons : soit en supprimant un module faisant disparaître de même tous les liens qui lui sont attachés, soit en définissant le chemin que l'utilisateur désire détruire par l'intermédiaire d'une fenêtre de dialogue spécifique. Ce chemin est alors défini en sélectionnant successivement le module et la variable d'entrée puis le module et la variable de sortie (Cf. figure 4.14).

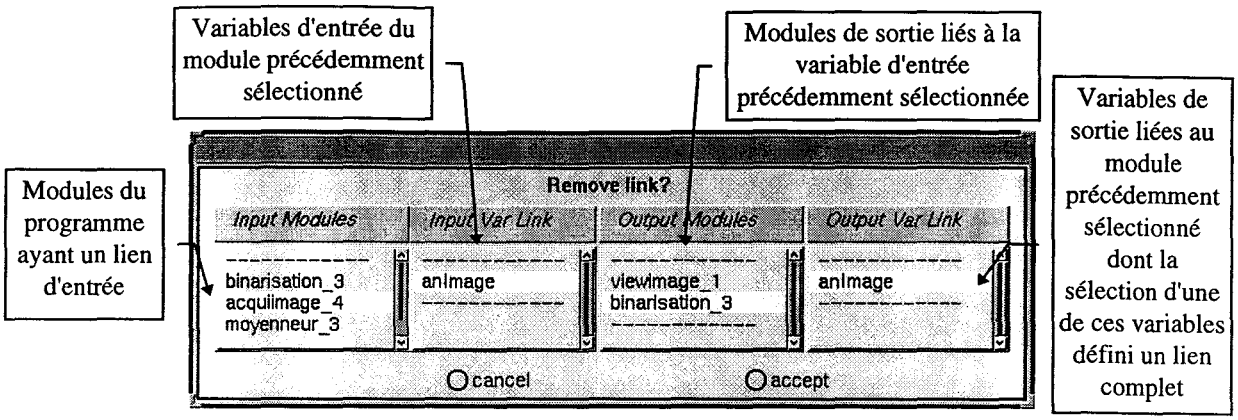


Figure 4.14 : *Fenêtre gérant les liens du programme modulaire.*

Parallèlement, les modules d'entrée et de sortie sont automatiquement sélectionnés dans l'interface de programmation visuelle ce qui permet à l'utilisateur de visualiser le chemin spécifié.

En tout état de cause, la construction de programmes sous forme modulaire est obtenue par agencement de modules autonomes, connectés interactivement par l'intermédiaire de la souris créant un réseau de relations simples et cohérentes entre les différentes données des modules employés.

IV.4.5 - Structures de Contrôle

Notre environnement peut être assimilé à un langage de programmation qui emploie une grammaire spatiale spécifique car elle nous permet de créer des programmes de manière visuelle [GOL 90]. Mais, pour être un véritable langage de programmation, notre système doit pouvoir diriger et fusionner le flux de données et manier les itérations et les boucles conditionnelles [AMB 90, WIL 91].

Pour ce faire, nous avons introduit des modules spécifiques du type : "For", "If", "While", "Merge", et "Switch", pour compléter ce langage. Ces modules peuvent contenir des expressions ou des variables, définies textuellement à leur création ou au cours de la simulation par l'utilisateur qui détermine les conditions d'évaluation de leur exécution (Cf. figure 4.15).

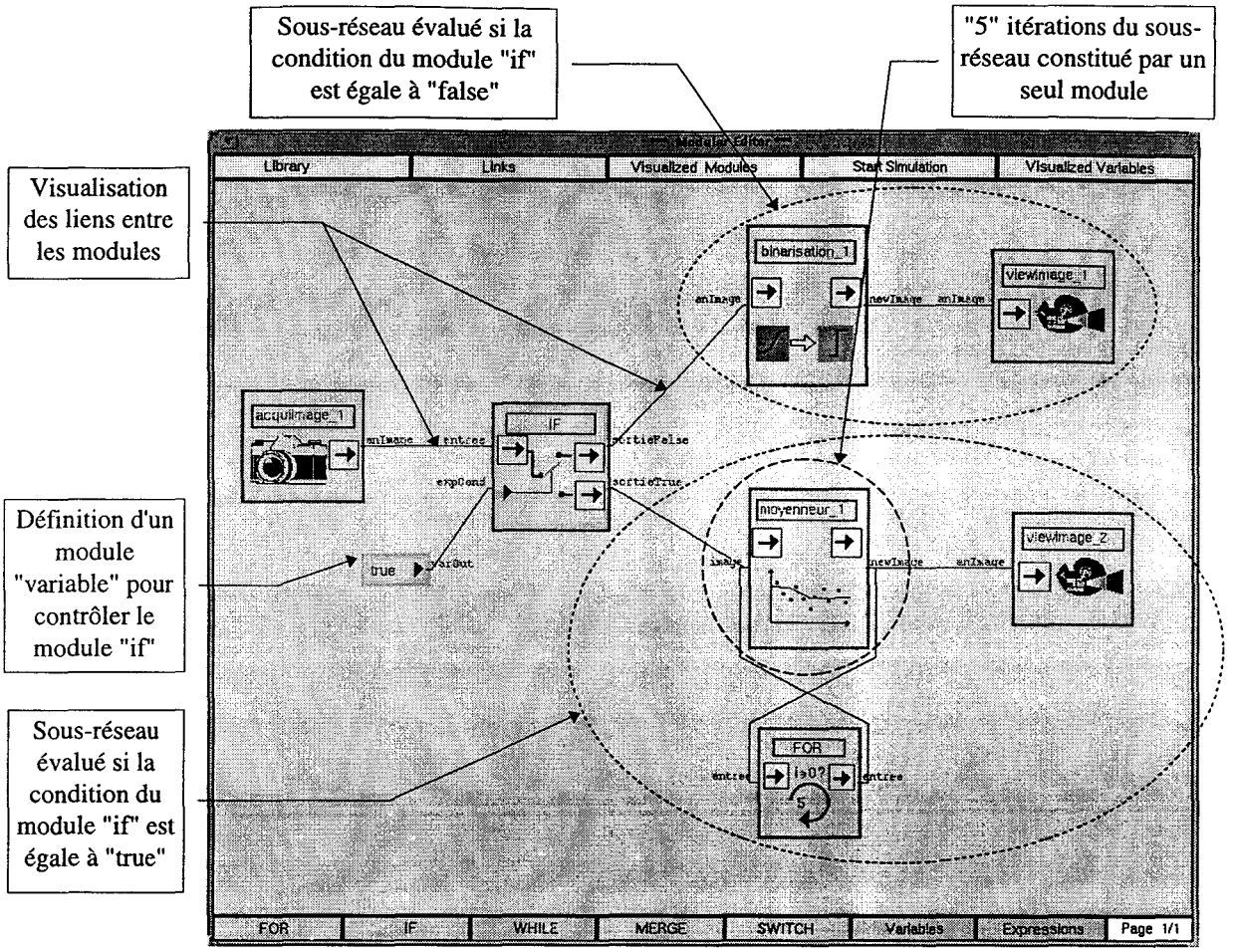


Figure 4.15 : *Un programme modulaire utilisant un module conditionnel et de boucle.*

Ces structures de contrôle sont employées pour déterminer le chemin par lequel les données circulent. Ainsi, nous trouvons, suivant le contrôle de flux ou leur fonctionnement particulier, deux sous-ensembles : les constructions boucles et les constructions conditionnelles :

- La construction boucle contrôle dans la programmation visuelle l'itération d'un sous-réseau jusqu'à ce que la condition de terminaison soit validée.

- La construction conditionnelle contrôle suivant la valeur de la condition du module, l'un ou l'autre des chemins correspondants au sous-réseau dépendant.

Ainsi, la première construction boucle accessible "For" itère les données qui circulent dans une section particulière du réseau un certain nombre de fois et la seconde "while" itère les données aussi longtemps qu'une condition particulière est rencontrée.

En ce qui concerne les constructions conditionnelles accessibles, nous trouvons "If/Else" qui dirigent les données qui circulent par un chemin si la condition est validée ou par un autre chemin dans le cas contraire. La construction "Merge" dirige le flux de données de deux chemins séparés sur un même chemin, si les données arrivent du premier, du second ou des deux chemins sans qu'aucune condition soit évaluée. Enfin, la construction "Switch sélectionne une des deux entrées utilisées par le réseau visuel suivant la valeur d'une condition.

Ces structures de contrôle ont le même emploi et le même comportement sémantique qu'un langage de programmation conventionnelles tel que le "C" ou le "Pascal".

IV.4.6 - Les variables et les expressions

Des variables peuvent être définies et des expressions utilisant ces variables peuvent alors être évaluées constituant dans la fenêtre de programmation visuelle des modules spécifiques. Une fois définies, ces variables et ces expressions peuvent être utilisées à la place d'entiers, de réels, d'arguments de modules, etc. Ainsi les expressions valides peuvent inclure des variables, des opérateurs arithmétiques et logiques, ou des constantes prédéfinies.

L'utilisation de ces variables et expressions est surtout requise par les structures de contrôle précédemment décrites (Cf. figure 4.15).

IV.4.7 - Mode d'exécution

L'évaluation des modules s'effectue dans notre environnement de manière séquentielle et aléatoire et s'obtient par l'intermédiaire d'un des boutons de la fenêtre de programmation visuelle (Cf. figure 4.12). L'ordre d'évaluation est en fait déterminé par la disponibilité des données d'entrées d'un module.

Plus explicitement, le moteur de la simulation recherche automatiquement le ou les modules ne possédant pas de données d'entrées et par la même ne pouvant avoir de liens définis avec ceux-ci pour les évaluer puis les modules étant en relation avec ses données de sorties. Si l'un de ces nouveaux modules peut avoir toutes ces données d'entrées disponibles, il sera alors évalué et ainsi de suite.

Comme il peut y avoir plusieurs modules susceptibles d'être traités au même instant le moteur de simulation va choisir, par simplicité, le premier parmi une liste répondant aux critères précédemment décrits. Ainsi, nous pouvons voir l'exécution d'un programme passer d'un sous-réseau à un autre sans que celui-ci ne soit obligatoirement entièrement évalué. Il faut donc que le moteur de la simulation sauvegarde l'état de toutes les données de sortie des différents modules.

IV.5 - COMPOSITION STRUCTURELLE D'UN PROGRAMME VISUEL

Afin de permettre la composition de larges et complexes programmes, une composition hiérarchique et structurelle a été établie. Elle va nous permettre d'homogénéiser la structure de la programmation visuelle et est conçue autour de deux concepts :

- l'abstraction procédurale et,
- la programmation par page.

Ces deux concepts reposent sur l'introduction d'un objet de programmation supplémentaire appelé "Objet composé" et peuvent être combinés pour créer une structure de programmation homogène.

IV.5.1 - L'Abstraction Procédurale

L'abstraction procédurale permet d'incorporer une forme de hiérarchie dans la programmation visuelle. Celle-ci constitue un module particulier dans notre environnement qui se caractérise par l'encapsulation de plusieurs modules en un simple opérateur. Ce concept est similaire à la notion de sous-programmes dans un langage de programmation textuelle. L'utilisation de ce type de module est principalement utile lors de composition de programmes importants en ne faisant apparaître dans l'environnement qu'un nombre limité de modules.

L'abstraction procédurale fait donc appel à un objet particulier qui permet d'englober plusieurs modules d'un sous-réseau du programme sous la forme d'une seule image. Cet objet, appelé "objet composé" est sous-classe des "Objets de Simulation" du modèle P.Os.T. et peut donc être structuré de la même manière. Plus explicitement, l'objet composé se substitue dans la structure du modèle P.Os.T. originel à l'objet de simulation et peut alors posséder une présentation et un traducteur, et ainsi une image fixe ou animée. La seule particularité de ce type d'objet est d'avoir une nouvelle variable d'instance contenant la liste des modules constituant une partie du programme visuel.

Un exemple de présentation d'une abstraction procédurale est présenté en figure 4.16.

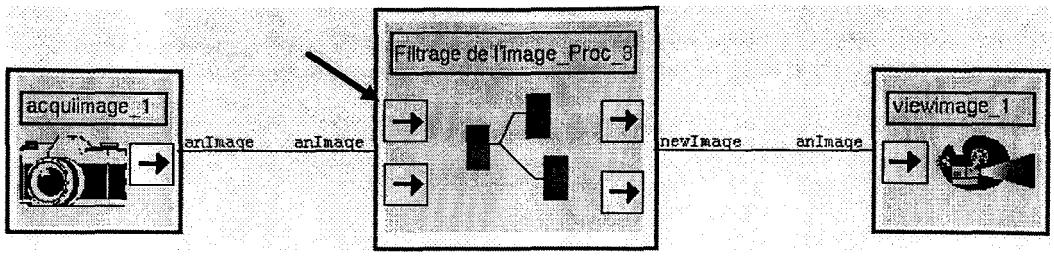


***Figure 4.16** : Visualisation d'une abstraction procédurale (désignée par la flèche) englobant plusieurs modules de filtrage d'image.*

Visuellement, les paramètres du réseau appartenant à la procédure paraîtront rattachés comme des données d'entrée et de sortie de ce nouveau module qui constitue alors les points d'entrée et de sortie de ce noeud. Le module qui représente l'abstraction procédurale sera vu par l'utilisateur comme relié de manière classique aux autres éléments du programme visuel.

Mais l'abstraction procédurale a bien d'autres intérêts que d'englober visuellement une partie d'un programme. Par exemple, elle facilite la réutilisabilité de certaines portions d'un réseau exécutant une fonction particulière, qui encapsulée peut alors être facilement rappelée plusieurs fois.

En outre, dès qu'un programme visuel a été conçu et sauvegardé, il peut être de nouveau employé dans un programme comme une abstraction procédurale.

Pour faciliter cette implantation, il est préférable d'encapsuler le programme global dès sa création dans un objet composé puisque celui-ci peut alors être considéré comme un module particulier. Cette organisation nous permet de restaurer toutes les informations d'un programme visuel et plus particulièrement les relations inter-modulaires si celles-ci sont sauvegardées par un seul et même objet constituant le programme, en l'occurrence l'objet composé.

Un programme visuel se présente comme un arbre hiérarchique homogène qui ne possède que des modules et des objets composés bâtis sur la même structure (Cf. figure 4.17).

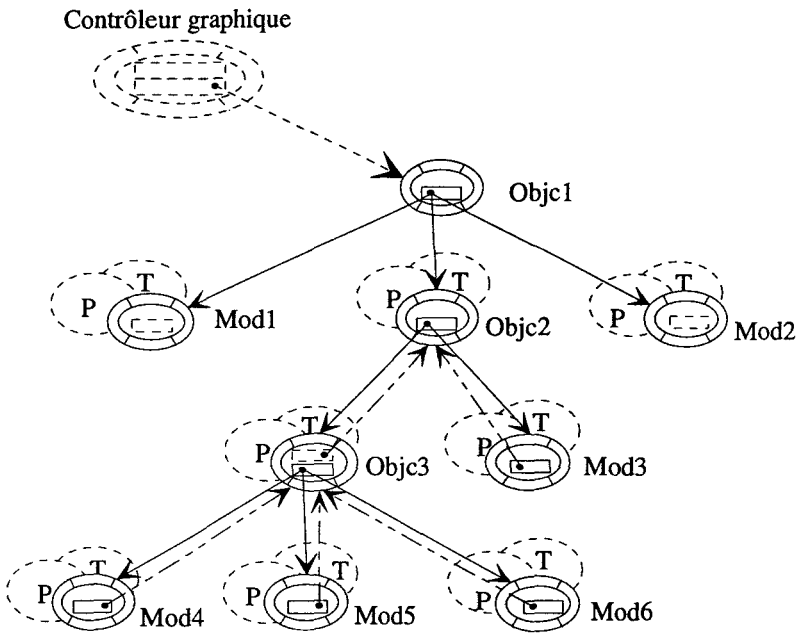


***Figure 4.17 :** Arbre hiérarchique d'un programme visuel employant l'abstraction procédurale*

Lors de l'exécution du programme le moteur de simulation ne prend en compte que les modules représentant les feuilles de l'arbre. Aussi pour permettre à celui-ci de connaître l'abstraction procédurale à laquelle appartient le module traité, nous avons établi une double liaison entre ces deux objets. Celle-ci nous permet de sélectionner lors de l'exécution d'un programme, le module évalué et les objets composés qui l'englobent si ceux-ci sont visualisés.

La création d'une abstraction procédurale s'effectue simplement en sélectionnant, par l'intermédiaire d'une boîte de dialogue, les modules que l'utilisateur veut encapsuler. Mais une nouvelle abstraction procédurale ne peut se faire que sur des modules qui appartiennent au même niveau de la hiérarchie (par exemple, si nous prenons l'arbre hiérachique de la figure 4.13 nous pouvons encapsuler le module "mod5" avec le module "mod6" mais pas avec le module "mod2").

D'autre part, pour permettre à l'utilisateur de visualiser le contenu d'une abstraction procédurale, une fenêtre basée sur la même architecture que la fenêtre de programmation visuelle (Cf. figure 4.7) a été créée et est présentée en figure 4.18.

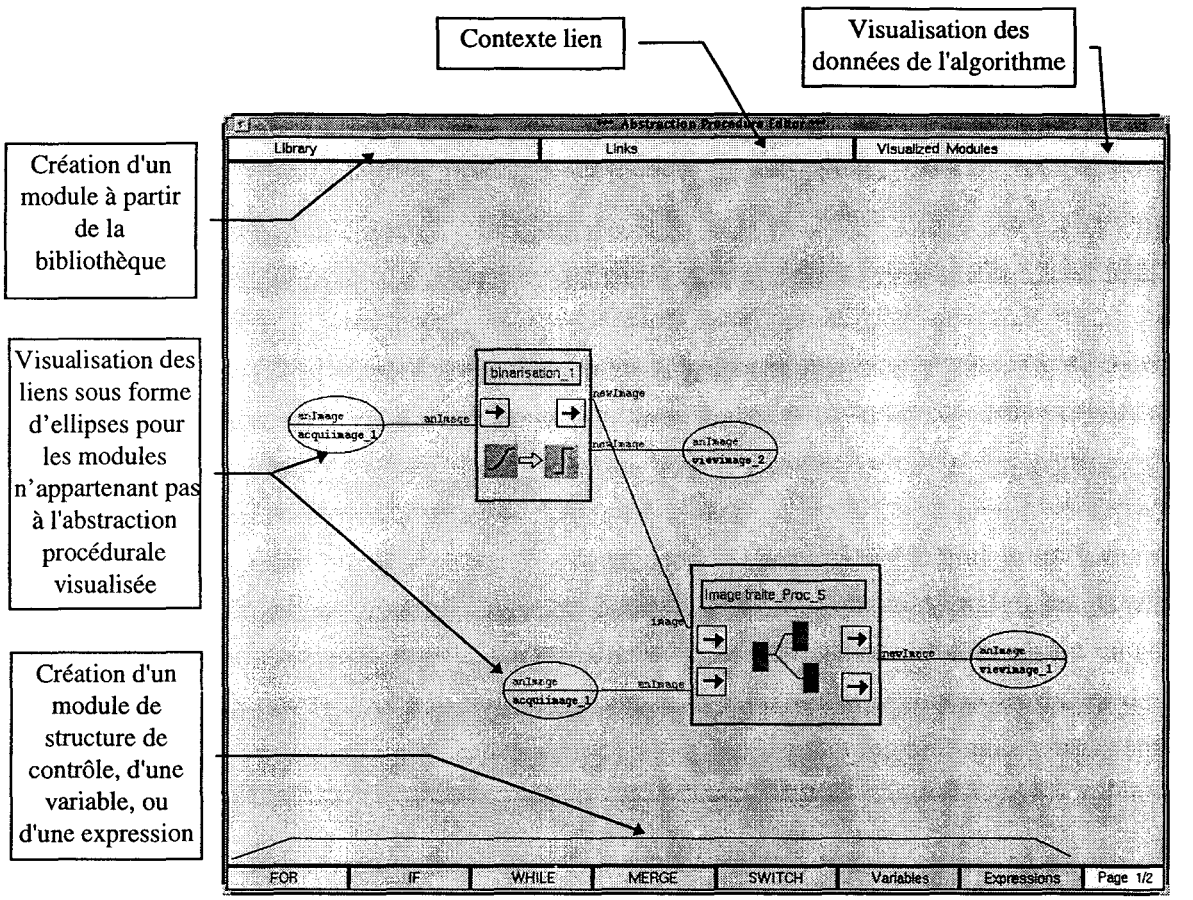


***Figure 4.18** : Fenêtre de visualisation d'une abstraction procédurale.*

Les données d'entrée et de sortie de l'abstraction procédurale ainsi visualisées sont représentées sous forme d'ellipses faisant apparaître le nom du module et de la donnée auquel elles se trouvent rattachées.

Cette fenêtre ne permet pas simplement de présenter les modules appartenant à une abstraction procédurale, elle peut également permettre d'étendre le sous-réseau visualisé en lui ajoutant de nouveaux modules et de nouvelles relations inter-modules. Ces relations peuvent être réalisées entre les modules de l'abstraction procédurale visualisée ou tout autre module du programme visuel. Pour ce faire, les contextes "lien" de toutes les fenêtres de programmation sont mis en relation par l'intermédiaire du concept de dépendance tel que, par exemple, la sélection d'un contexte lien dans une fenêtre d'abstraction procédurale oblige automatiquement toutes les autres fenêtres visualisant le même programme à adopter celui-ci.

Cependant, si elles ont une certaine autonomie, ces fenêtres n'ont lieu de se trouver dans l'environnement que pendant l'évaluation du programme auquel appartient l'abstraction

procédurale visualisée. Aussi, quand l'utilisateur ferme la fenêtre de programmation visuelle ou change de programme visuel, toutes les fenêtres visualisant une abstraction procédurale sont fermées automatiquement et ne laissent dans l'espace de travail aucune information obsolète.

IV.5.2 - La programmation par page

Pour aider l'utilisateur à construire de volumineux programmes sans nécessairement faire appel au concept d'abstraction procédurale, nous avons introduit les structures pour composer ceux-ci sur plusieurs pages.

Comme nous venons de le voir, un objet composé englobe l'ensemble du programme et constitue en fait sa page initiale. Aussi, ajouter une nouvelle page revient à définir un nouvel objet composé pour celle-ci.

Si nous excluons dans un programme visuel les modules d'abstraction procédurale et que nous le programmons sur plusieurs pages, nous pouvons le représenter également sous la forme d'un arbre hiérarchique (Cf. figure 4.19).

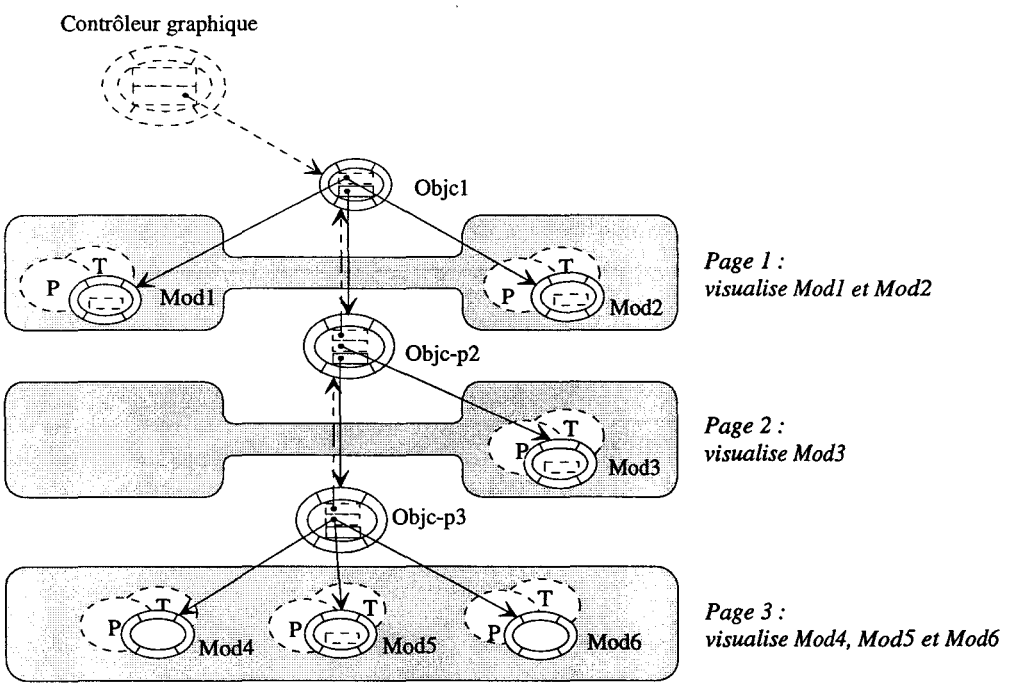


***Figure 4.19 :** Arbre hiérarchique d'un programme visuel programmé sur plusieurs pages.*

La double liaison entre les objets composés constituant ces pages permet à l'utilisateur de naviguer entre celles-ci.

D'autre part, pour l'aider à se repérer dans la hiérarchie, la double information constituée du nombre de pages appartenant au programme visuel et du numéro de la page visualisée apparaît au bas de la fenêtre de visualisation.

IV.5.3 - La visualisation structurelle d'un programme

Le fait d'avoir considéré le noeud d'une page comme un objet composé renforce encore les possibilités de hiérarchisation, car nous pouvons considérer que chaque module d'abstraction procédurale étant un objet composé constitue une page initiale. De ce fait, un module d'abstraction procédurale peut également être composé de plusieurs pages.

L'utilisation dans un programme visuel du concept d'abstraction procédurale et de la programmation par page aboutie à l'obtention d'un programme qui peut être schématisé sous la forme d'un arbre vertical (pour l'abstraction procédurale) et horizontal (pour les pages) dont un exemple est présenté en figure 4.20.

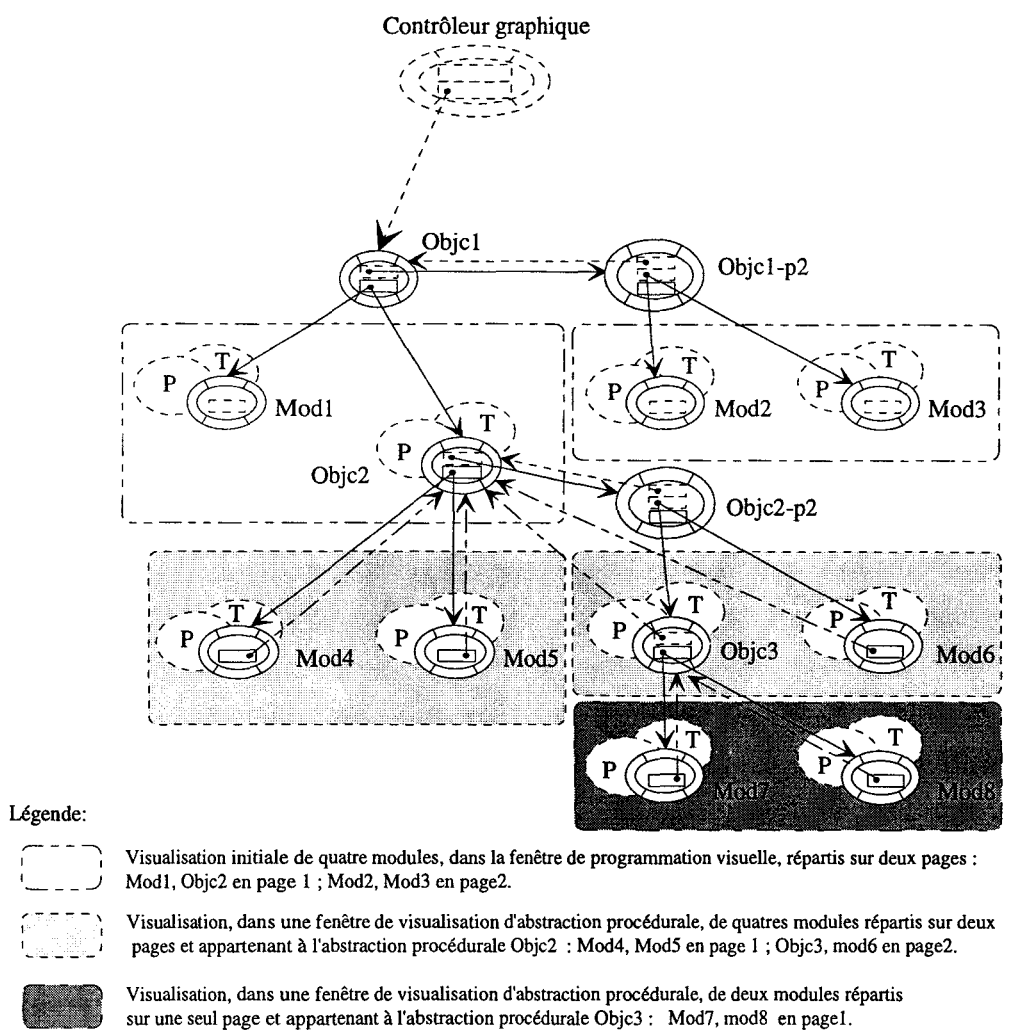


***Figure 4.20 :** Représentation structurelle d'un programme visuel.*

La réunion de ces deux concepts ne limite plus le nombre de modules dans un programme, si ce n'est la place mémoire pour créer dans l'environnement tous les objets nécessaires à son élaboration.

Cependant, si ces deux notions de programmation sont utilisées de manière irrationnelle, elles ont tendance à rendre un programme illisible. Par exemple, si nous prenons l'arbre hiérarchique de la figure ci-dessus, l'utilisateur doit ouvrir trois fenêtres lui permettant de regarder cinq contextes visuels différents pour visualiser tous les modules.

Aussi l'utilisateur peut obtenir une aide pour se retrouver dans un tel programme en ouvrant une fenêtre qui lui permet d'inspecter la structure hiérarchique et les liens définis (Cf. figure 4.21).

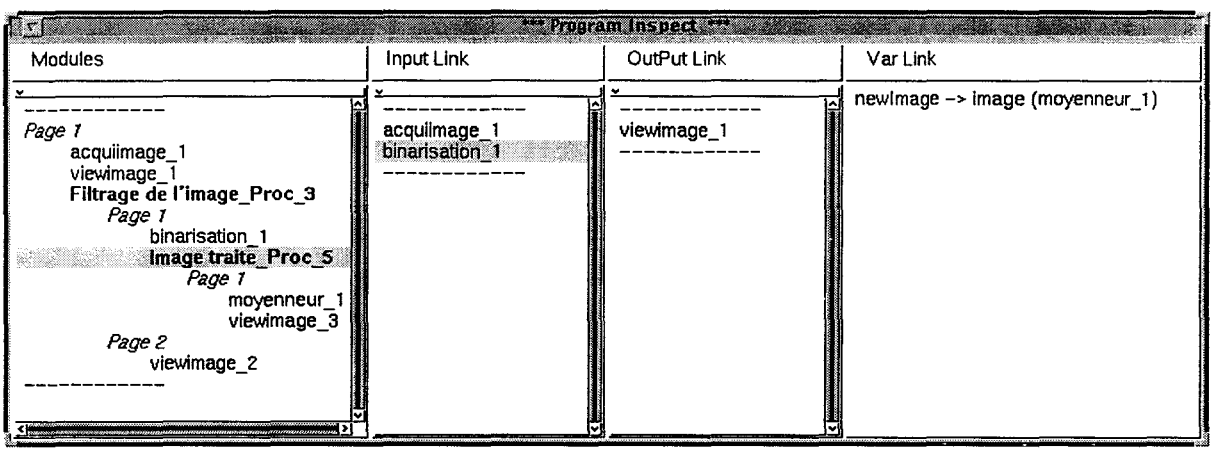


Figure 4.21 : *Inspection hiérarchique d'un programme visuel.*

La structure modulaire du programme est visualisée dans la première sous-vue faisant apparaître le contenu de toutes les pages et de toutes les abstractions procédurales sous la forme d'une liste hiérarchique. Pour faciliter la lecture d'un tel arbre les différentes branches correspondant aux abstractions procédurales et aux différentes pages qui le constituent se retrouvent successivement décalées et les pages sont visualisées en italique et les abstractions procédurales en gras.

Lorsque l'utilisateur sélectionne un des éléments de cette liste, les modules dont une entrée ou une sortie est en relation avec celui-ci, apparaissent respectivement dans l'une ou l'autre des deux sous-vues adjacentes. La sélection d'un module d'entrée ou de sortie affiche dans la dernière sous-vue les liens entre les deux composants sélectionnés caractérisés par une donnée d'entrée et une donnée de sortie. Si la donnée d'entrée ou de sortie correspond à un module de l'abstraction procédurale que l'on est en train d'inspecter, nous affichons cette information, entre parenthèse, à côté de la variable concernée.

L'inspection hiérarchique d'un programme peut être effectuée sur l'ensemble du programme visuel ou simplement se limiter à la branche d'une abstraction procédurale.

IV.6 - VISUALISATION DES INFORMATIONS DES MODULES

Lors de l'exécution de l'algorithme créé visuellement, l'utilisateur peut également, suivant son choix, visualiser plus explicitement le traitement d'un ou plusieurs des modules employés ainsi que chacune des données constituant ce nouveau programme dans des fenêtres autonomes.

Ainsi la visualisation de l'exécution d'un ou plusieurs modules est obtenue en plusieurs étapes. La première consiste à mémoriser les modules que l'utilisateur souhaite visualiser et dont la sélection s'effectue par l'intermédiaire de la souris et une fenêtre de dialogue spécifique présentant la liste complète des modules du programme visuel (Cf. figure 4.22). Ce choix s'accompagne, comme dans la gestion des liens, de la sélection automatique des différents modules dans l'espace de programmation visuelle permettant à l'utilisateur de contrôler les modules effectivement sélectionnés.

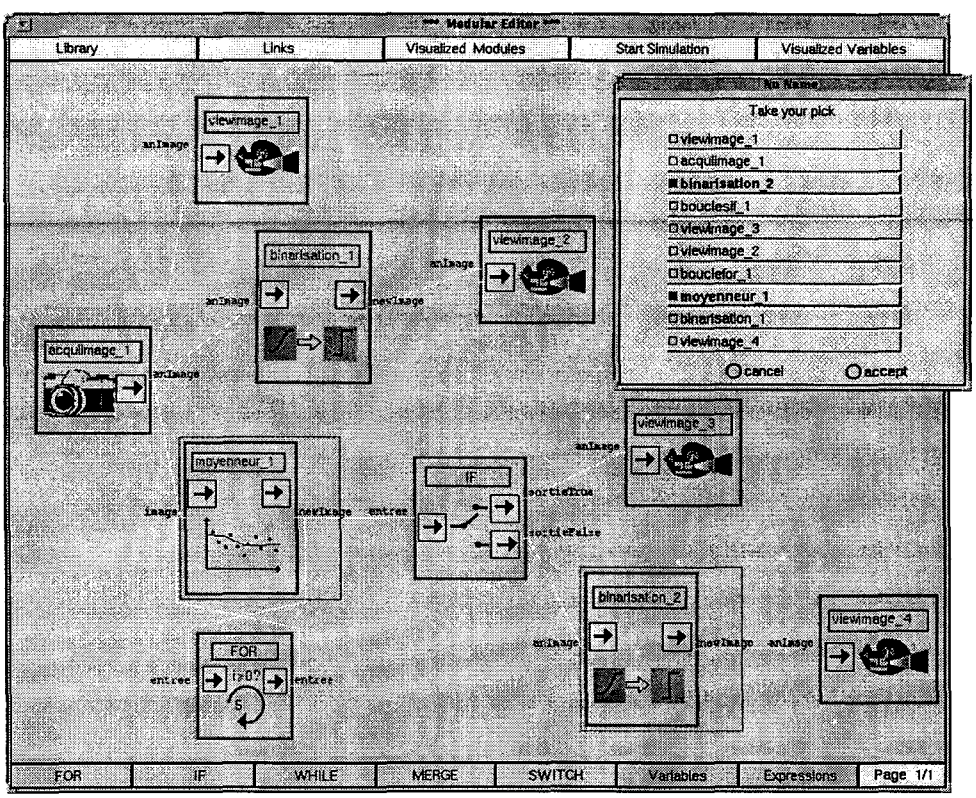


***Figure 4.22** : Sélection des modules visualisés.*

Ainsi lors de l'évaluation des différents modules qui constituent le nouveau programme nous ouvrons automatiquement une fenêtre de visualisation contenant le code du module que nous avons précédemment sélectionné.

Cette fenêtre est de même type que celle utilisée pour visualiser le traitement d'un algorithme dans la visualisation de programme (Cf. chapitre III.4) et est gérée avec les mêmes concepts.

Ainsi, nous pouvons faire appel aux différents outils que nous avons développés pour l'animation d'algorithmes, permettant de présenter (Cf. figure 4.23) :

- l'algorithme traité sous sa forme textuelle (écrit en Smalltalk ou en Pascal) avec affichage des expressions en cours d'évaluation,

- le comportement global ou de synthèse du module et,

- les structures de données ou les expressions (que l'utilisateur jugera nécessaires à sa compréhension) sous une forme graphique et animée.

Cependant, l'ouverture dans l'environnement de programmation visuelle de cette fenêtre a pour conséquence d'interrompre le processus d'évaluation des autres modules. Aussi, après avoir visualisé les informations qu'il désirait inspecter, l'utilisateur peut en fermant la fenêtre de visualisation d'algorithme soit interrompre le processus d'exécution soit l'obliger à reprendre le traitement dans le contexte où il se trouve.

D'autre part, pour ne pas contraindre l'utilisateur à ouvrir systématiquement une fenêtre de visualisation d'algorithme pour visualiser les données des modules qui constituent le nouveau programme, celles-ci peuvent être obtenues par l'intermédiaire d'un des boutons du menu général de la plate-forme de programmation. Ainsi, en sélectionnant une ou plusieurs des données qui appartiennent aux différents modules du programme, nous ouvrons des fenêtres autonomes permettant de visualiser leurs valeurs.

Ces fenêtres sont de même type que celles précédemment décrites pour présenter les structures de données dans la visualisation de programmes (Cf. chapitre III.5).

Par contre, la remise à jour de leur représentation lançant éventuellement l'animation est effectuée automatiquement en conservant une relation de dépendance non plus avec la fenêtre présentant l'algorithme traité mais avec la fenêtre de programmation visuelle.

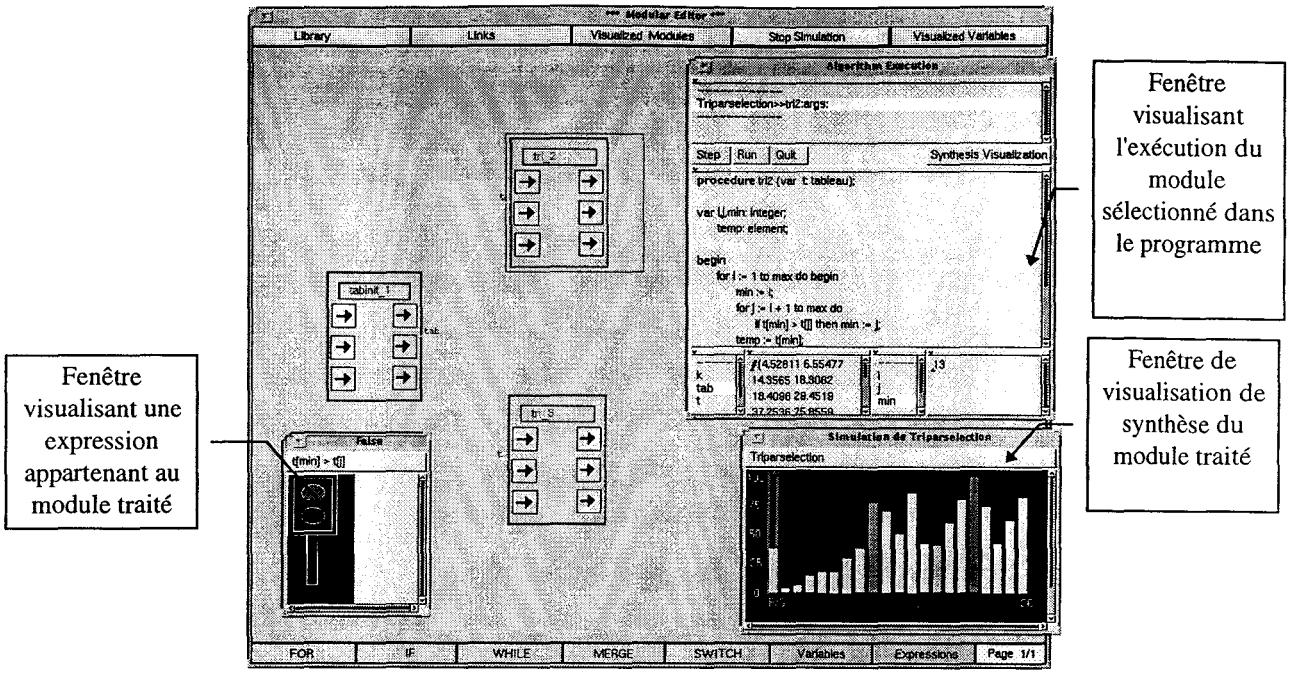


Figure 4.23 : *Visualisation de quelques informations d'un module [1] dans différentes fenêtres autonomes.*

IV.7 - UNE PROGRAMMATION VISUELLE INTERACTIVE

Une autre originalité de notre simulation est de pouvoir changer le code d'un des modules indépendamment de l'algorithme originel ou de tout autre module de même nature pour tester différentes variantes. Ainsi, lorsqu'un seul ou divers programmes visuels font appel plusieurs fois au même algorithme, les modules créés peuvent avoir un code sensiblement différent et donc un comportement propre.

Pour ce faire, lors de la création du module nous copions le code de l'algorithme originel constituant un clone que nous compilons sous un nouveau nom (Cf. figure 4.24).

Celui-ci constituant un nouveau message du programme originel sera l'algorithme effectivement évalué lors de l'exécution du module. Ainsi, si le code de ce clone est identique à l'algorithme originel lors de sa création, il peut être modifié à loisir par l'utilisateur sans changer le code des autres modules de même nature dans l'environnement.

[1] Ce module fait partie d'un programme qui traite selon deux méthodes de tri un tableau initialisé aléatoirement.

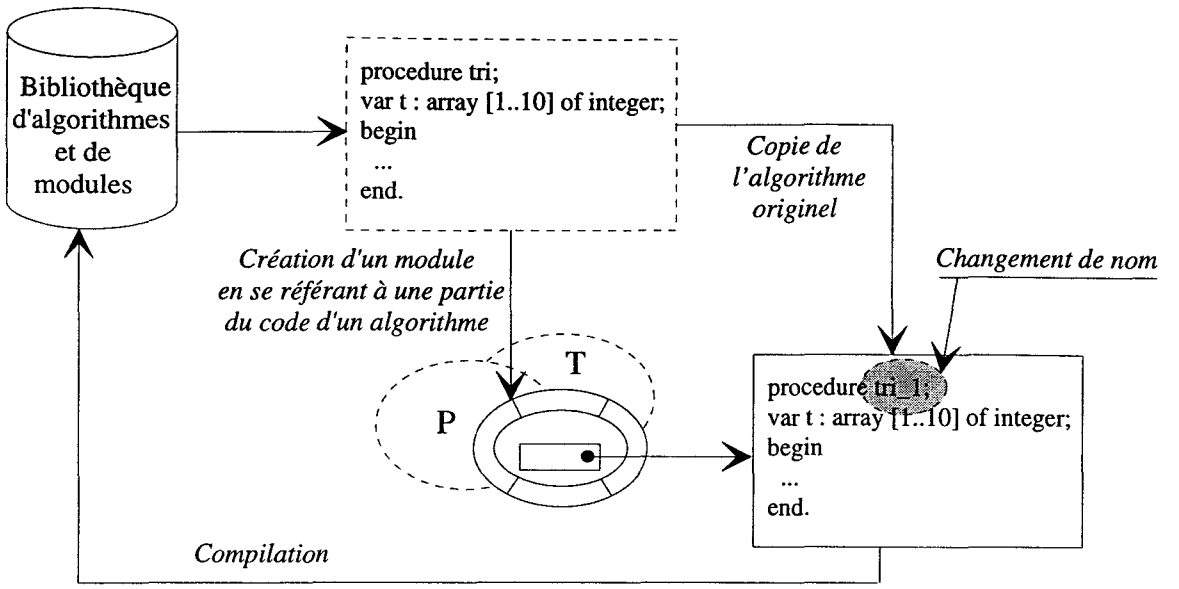


Figure 4.24 : *Création d'un module et gestion de son code*

Nous avons d'autre part délibérément fait le choix que ce nouveau message ne soit pas considéré comme un nouvel algorithme s'intégrant à la bibliothèque globale pour ne pas l'encombrer de messages inutiles. La conséquence de ce choix est que le module est chargé de gérer son propre code, ainsi que toutes les modifications qui lui seront apportées en le sauvegardant. Toutes ces altérations ne peuvent être alors considérées que par le module défini dans le programme visuel auquel il appartient.

D'autre part, sa destruction s'accompagne dans l'environnement de la destruction du message compilé constituant le clone et tout chargement de programmes préalablement sauvegardés régénère automatiquement le dernier code connu des modules qui le composent.

L'architecture adoptée et les concepts mis en oeuvre nous permettent ainsi de conserver un environnement interactif et convivial [VAN 96a] qui accepte, quel que soit le type d'algorithme visualisé de :

- pouvoir changer le code source d'une méthode ou d'une procédure au cours de l'exécution ou de la simulation graphique de cet algorithme,

- pouvoir agir sur une structure de données du programme, c'est-à-dire changer la valeur d'une donnée de l'algorithme par l'intermédiaire des deux inspecteurs associés à la fenêtre présentant l'algorithme traité. Ce changement de valeur entraîne également automatiquement une remise à jour de l'image graphique de cette donnée, si celle-ci est visualisée.

- visualiser dans des fenêtres autonomes d'animation de données la valeur d'une expression sélectionnée à la souris dans le texte source ou définie textuellement par l'utilisateur.

IV.8 - CONCLUSION

L'originalité de la plate-forme de programmation visuelle que nous avons conçue est l'obtention d'un environnement de programmation homogène, convivial et interactif basé sur l'emploi d'un langage naturellement visuel et facilement compréhensible. L'outil permet ainsi à n'importe quel utilisateur de créer à partir d'une bibliothèque de modules de nouveaux programmes en assemblant de manière graphique et interactive différents composants logiciels.

L'emploi d'objets visuels de programmation accorde la flexibilité et la réutilisabilité nécessaires pour traiter différents problèmes et proposer des prototypes de résolution.

Notre environnement permet également de visualiser chacune des variables d'un programme ainsi que l'exécution des modules qui le compose indépendamment les uns des autres. Cette visualisation s'effectue sous la forme d'une animation graphique faisant simplement appel aux outils développés pour l'animation d'algorithme et au concept d'architecture d'animation utilisé par celle-ci : remise à jour des présentations par un lien de dépendance, gestion de l'écran sous la forme de multi-fenêtres, etc.

En outre, l'outil de programmation visuelle est réalisé en employant des concepts permettant à un utilisateur d'altérer à n'importe quel instant, de façon simple et conviviale, les différentes informations composant les modules d'un programme : valeur des données, code source, etc. Ces modifications sont réalisées en conservant une cohérence entre les informations visualisées et l'état du programme n'obligeant pas nécessairement l'utilisateur à réexécuter tout un programme pour visualiser les effets induits de ces changements.

D'autre part, bénéficiant du concept d'abstraction procédurale qui encapsule une partie d'un programme dans un module autonome, de la programmation sur plusieurs pages, alliés aux différents outils de visualisation développés, l'utilisateur peut écrire de très larges programmes en ne regardant que l'exécution d'une partie de ceux-ci. Par conséquent, nous pouvons pallier le défaut majeur des visualisations d'algorithmes c'est-à-dire le fait que les représentations des différentes informations et surtout du comportement global ne peuvent être généralement établies que pour des programmes réduits.

Ainsi, en combinant les outils de visualisation de programme et de programmation visuelle, nous obtenons un environnement visuel homogène particulièrement appréciable dans le cas des traitements scientifiques.

Dans le prochain chapitre, nous allons voir l'intérêt d'introduire ces outils dans un environnement de programmation classique ou spécialisé, dans un but d'éducation, d'aide au développement ou de recherche en prenant quelques exemples concrets empruntés à l'informatique, au traitement d'image et à l'automatique classique.

CHAPITRE V

VALIDATION DU SYSTÈME DE VISUALISATION ET D'ANIMATION D'ALGORITHMES PROPOSÉ

V.1 - VALIDATION DU SYSTEME PROPOSE

Ayant décrit les méthodes et les concepts qui nous permettent de visualiser les informations d'un programme dans un environnement convivial et interactif, nous allons évaluer et valider le système proposé dans un contexte expérimental. Pour ce faire nous présentons dans ce chapitre quelques unes des représentations et animations réalisées en démontrant leur généricité.

Il existe actuellement de nombreuses méthodes permettant d'évaluer la qualité d'un logiciel. La figure 5.1 présente pour les interfaces homme-machine la synthèse des contextes d'évaluation proposée par Senach [SEN 90a, 90b].

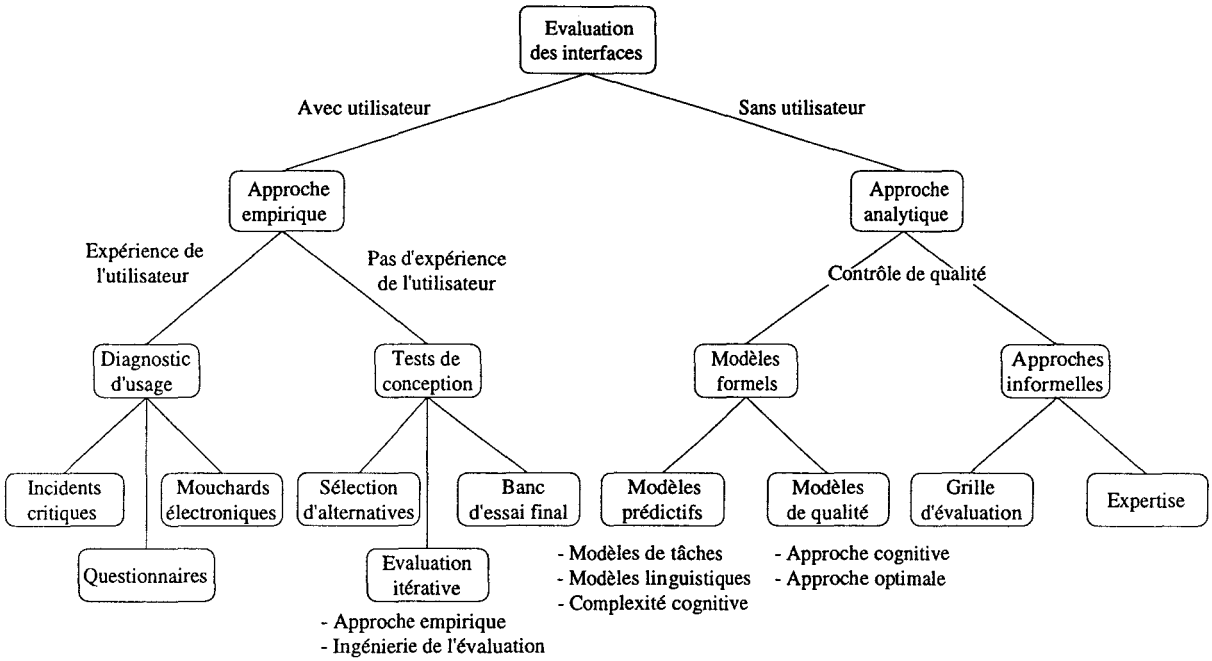


***Figure 5.1** : Classification des principaux contextes d'évaluation*

N'étant encore qu'en phase de prototypage, l'évaluation du système de visualisation et d'animation d'algorithmes proposé ne peut se faire que selon une approche analytique qui vise à contrôler sa qualité selon un modèle défini a priori dit informel ou formel.

L'approche informelle est pour nous la plus séduisante et la plus facile à mettre en oeuvre car elle permet dans le cas d'une expertise de mettre l'analyseur pour un temps à la place de l'utilisateur. Cette méthode présente cependant le défaut que l'analyseur n'est pas toujours représentatif et objectif. Néanmoins ayant considéré que cette application s'adressait aussi bien aux enseignants qu'aux chercheurs nous pouvons nous considérer comme un utilisateur potentiel apte à effectuer une première évaluation du logiciel en l'occurrence nous simulerons l'utilisateur professionnel (Cf. chapitre II.2).

V.1.1 - Le protocole de validation

Comme toute validation nous faisons appel à un protocole chargé de définir les paramètres expérimentaux à évaluer. Ceux-ci s'appuies sur les critères ergonomiques énoncés au premier chapitre et sont :

- la conservation de l'homogénéité du système au niveau structurelle et visuelle,
- le niveau de la charge de travail (facilité et temps de conception d'une animation),
- la flexibilité de l'interface pour différents niveaux d'expérience de l'utilisateur,
- le niveau d'interactivité.

Auxquels nous pouvons également ajouter un paramètre important qui est :

- le niveau d'indépendance du code de l'algorithme par rapport à son animation.

Le protocole d'évaluation établi, nous pouvons passer à la description du cadre expérimental chargée de mettre en exergue pour un certain nombre d'exemple d'animation les points forts et les points faibles de l'application.

V.1.2 - Cadre expérimental

La validation du système proposé est effectuée en animant quatre algorithmiques choisis pour leur représentativité et leur domaine d'application différent.

Le premier algorithme présenté est un programme de tri par sélection écrit en employant la syntaxe du langage Smalltalk. Quant au second, il est implémenté en Pascal et concerne la recherche des circuits hamiltoniens d'un graphe.

Ces deux premiers exemples, étant des algorithmes élémentaires, sont visualisés en employant plus spécifiquement les outils de visualisation de programmes présentés au troisième chapitre. En fait, nous allons de manière non exhaustive décrire les possibilités d'une partie de notre environnement à travers la présentation des animations des informations les plus représentatives de ces programmes.

Nous constatons que lors de leur création, nous utilisons très souvent la couleur de fond de l'image pour rendre invisible les différents éléments graphiques qui la composent. Ceci est dut au fait que toutes les représentations créées peuvent être celles d'un composant logiciel de la programmation visuelle et que celui-ci utilise déjà le paramètre de visibilité adjoint à chacun des objets graphiques définis pour permettre de gérer l'affichage ou non d'un module lors de la navigation de l'utilisateur entre les différentes pages du programme.

Le troisième algorithme présenté est un programme de traitement d'image créé et visualisé par l'intermédiaire de l'outil de programmation visuelle. Cet exemple élémentaire nous permet de visualiser les effets de différents filtres ayant pour vocation de détecter les contours d'une image et met en exergue les facilités de combinaisons et d'associations modulaires d'un tel environnement.

Enfin le quatrième exemple est un algorithme permettant de visualiser l'asservissement d'une antenne en position et en poursuite modélisé sous la forme d'une fonction de transfert.

Les exemples choisis ne décrivent pas toutes les animations développées et les possibilités de notre environnement. Aussi, à défaut d'être exhaustif, nous avons avant tout préféré montrer l'aspect pluridisciplinaire et les avantages que peut avoir un tel environnement dans l'enseignement comme dans la recherche, même si les exemples traités sont relativement simples.

V.2 - TRI PAR SELECTION

Le premier exemple présenté concerne donc le tri d'un tableau. Concrètement, l'action de trier ou d'ordonner une liste d'objets selon une relation d'ordre linéaire donnée est tellement importante et fréquente, quel que soit le domaine scientifique considéré, que le sujet a donné lieu à de multiples élaborations d'algorithmes.

Nous nous proposons d'animer un tri de tableau relativement simple appelé "tri par sélection" et qui est choisi généralement pour ordonner un petit ensemble d'objets [SED 83].

V.2.1 - Une solution algorithmique et programmatique

Le principe de l'algorithme d'un tri de tableau par sélection est de le construire en déterminant l'élément "minimal" ou de plus petite valeur pour l'échanger avec celui en première position du tableau non encore trié. Pour effectuer cette permutation il suffit donc de déterminer l'indice de l'élément minimal du tableau.

A la première itération ou passe, l'algorithme consiste donc à trouver celui-ci en le parcourant du plus petit indice au plus grand et en comparant l'élément courant avec l'élément minimal déjà rencontré. Après avoir été déterminé, il ne suffit plus que d'échanger cet élément minimal avec l'élément ayant la première position. A l'étape suivante, on recherche l'indice de l'élément qui possède la deuxième plus petite valeur pour l'échanger avec l'élément qui a la deuxième position, et l'on continue ainsi jusqu'à ce que le tableau soit entièrement trié.

Si nous généralisons l'algorithme, au $i^{ème}$ passage, on recherche l'indice de l'élément minimum en parcourant le tableau de i à n, (n étant le nombre d'éléments de celui-ci) pour l'échanger avec l'élément d'indice i. D'autre part, au $i^{ème}$ passage, les i premiers éléments sont triés et les $n - i$ éléments restants sont forcément supérieurs. Le tri peut donc s'arrêter à la $n - 1^{ème}$ itérations, en effet, l'élément à la $n^{ème}$ position sera forcément à la sienne.

L'algorithme de tri par sélection est implanté dans l'environnement sous la forme de deux messages appelés "triparelection" et "triparselec:arg:" appartenant à la classe "Triparselection" et est présenté en figure 5.2.

```
triparselection
| tab rand nb |

    nb := 20.
    tab := Array new: nb.
    rand := Random new.
    1 to: nb do: [:k | tab at: k put: (rand next * 100)].
    tab := self triparselec: tab arg: nb

triparselect: array arg: n
    | min temp |
    1 to: (n - 1)
      do: [:i | min := i.
            (i + 1) to: n
                  do: [:j | (array at: min) > (array at: j)
                                          ifTrue: [min := j]].
            temp := array at: min.
            array at: min put: (array at: i).
            array at: i put: temp]
    ^array
```

Figure 5.2 : *Deux messages Smalltalk implémentant l'algorithme de tri par sélection.*

Ceux-ci correspondent respectivement à la partie initialisation du tableau et à l'algorithme de tri proprement dit, ce qui permet à l'outil de visualisation de programme de réutiliser ce dernier comme module algorithmique. Plusieurs choix ont été fait dès l'implantation de l'algorithme ou imposés par des contraintes du langage (un tableau en Smalltalk commence toujours avec comme indice initial "1").

Mais en dehors de ces problèmes d'implémentation et de généricité du programme, nous allons voir que la visualisation et l'animation des informations qu'il comporte peuvent être gérés indépendamment de ces différents choix.

V.2.2 - Présentation et animation des informations du programme.

La première information importante qui permet de visualiser toutes les informations ultérieures est la présentation automatique du traitement du programme dans une fenêtre spécialisée dont les fonctionnalités internes et externes sont plus explicitement présentées au chapitre III.4 (Cf. figure 5.3).

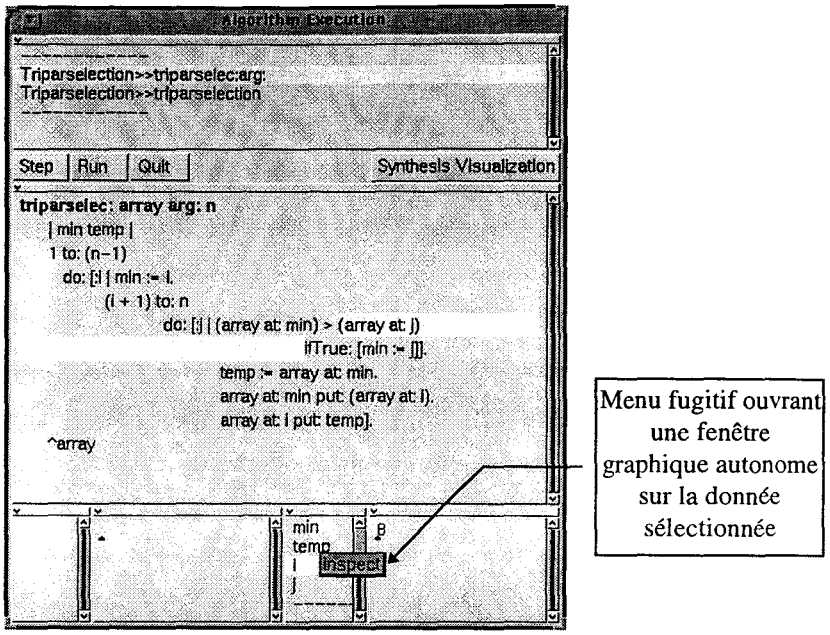


Figure 5.3 : Visualisation de l'exécution d'un algorithme

Cette visualisation nous permet de présenter le code de l'algorithme de tri ainsi que l'état de contrôle caractérisé par la visualisation de l'expression évaluée et du message exécuté.

Cette fenêtre nous permet également de présenter de manière textuelle, par l'intermédiaire des deux inspecteurs qui lui sont attachés, la valeur d'une des données que l'utilisateur aura sélectionnée.

Une des fonctions du menu fugitif associé à cette sous-vue permet également la visualisation de cette donnée de manière graphique dans une fenêtre autonome.

V.2.2.1 - La visualisation de quelques données

A chaque donnée ou objet caractéristique de l'environnement, nous avons joint une représentation et une animation correspondant à leurs comportements. Plus spécifiquement, un objet P.Os.T. servant de modèle d'animation leur a été associé à l'aide de l'éditeur graphique et de l'outil de prototypage présenté au chapitre III.7.2.2.

Dès que la fenêtre de visualisation d'une donnée est ouverte, celle-ci fait appel au modèle P.Os.T par l'intermédiaire d'une méthode d'affichage polysémique implantée dans chacune des classes des données considérées. De ce fait, l'utilisateur ne connaît pas a priori de manière univoque le code exécuté et donc l'image appelée, cela dépend en fait de l'objet receveur du message et donc de la donnée inspectée. Cette méthode est également implantée dans la classe au sommet de la hiérarchie du langage Smalltalk ("Object") ce qui permet alors à n'importe quel objet ne possédant pas son propre message d'affichage de présenter au moins sous forme textuelle son contenu.

L'utilisateur peut alors visualiser toutes les données du programme en demandant à un des deux inspecteurs de présenter celles-ci de manière graphique ; par exemple, visualiser l'état de la variable "array".

V.2.2.1.1 - Présentation et animation d'une donnée de type tableau

Pour visualiser de manière graphique une donnée de type tableau sans que l'utilisateur ait à définir une quelconque représentation, un modèle P.Os.T. générique, dont la représentation correspond à un graphe à barres, a donc été associé à une variable de classe de la classe "Array" de Smalltalk (Cf. figure 5.4).

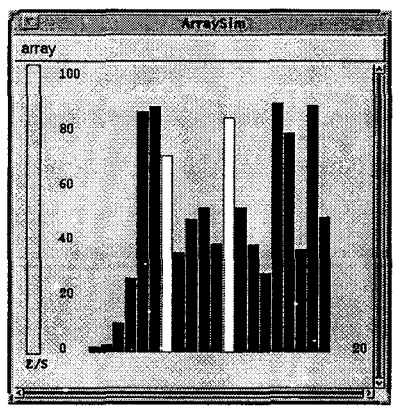


***Figure 5.4** : Visualisation de la donnée "array" du tri par sélection*

Chaque élément du tableau est ainsi représenté sous la forme d'un rectangle dont la hauteur est proportionnelle à la valeur de la donnée qu'elle représente.

Dans le programme implémenté (cf. figure 5.2), le tableau a été initialisé avec 20 éléments et des valeurs aléatoires comprises entre 0 et 100, mais il aurait pu être initialisé avec plus d'éléments et d'autres objets, par exemple, des caractères. La création d'une animation de données doit alors être suffisamment générique et polymorphe pour prendre en compte ces éventuels changements de paramètres en établissant une représentation et une animation adéquates. Ceci est d'autant plus critique que la visualisation créée est la même pour n'importe quelle définition de tableau et que nous permettons et incitons l'utilisateur à modifier les données ainsi que le code du programme.

Ce problème a donc dut être pris en compte dans l'élaboration de la représentation du tableau et de ces éléments en particulier.

Nous avons remarqué que, quelle que soit la nature des objets et de l'algorithme considéré, nous différencions les éléments qu'il comporte suivant un seul critère. Celui-ci peut

alors correspondre à une unique variation de donnée et à un seul changement de paramètre graphique, en l'occurrence, la hauteur des différents rectangles.

Cette variation de hauteur va ainsi constituer la première animation que nous allons adjoindre systématiquement à tous les rectangles de la représentation pour qu'ils soient représentatifs de la valeur des éléments du tableau qu'ils symbolisent. D'autre part, comme nous désirons que leur affichage se fasse toujours en essayant de trouver une visualisation optimale, c'est-à-dire en employant si possible tout l'espace de visualisation qui leur est réservé, nous avons ajouté une contrainte à cette animation. Celle-ci est alors chargée d'afficher l'objet graphique proportionnellement à la plus grande valeur du tableau déterminée par l'objet de simulation du modèle P.Os.T.

Mais pour que notre représentation du tableau soit véritablement polymorphe, il faut également gérer le changement du nombre d'éléments. Il convient alors d'adjoindre à la présentation une méthode permettant de faire varier visuellement le nombre de rectangles créés et donc d'éléments. Ceci est réalisé en adoptant une représentation initiale adaptée (Cf. figure 5.5) et en donnant au traducteur des méthodes d'animations spécifiques (Cf. figure 5.6).

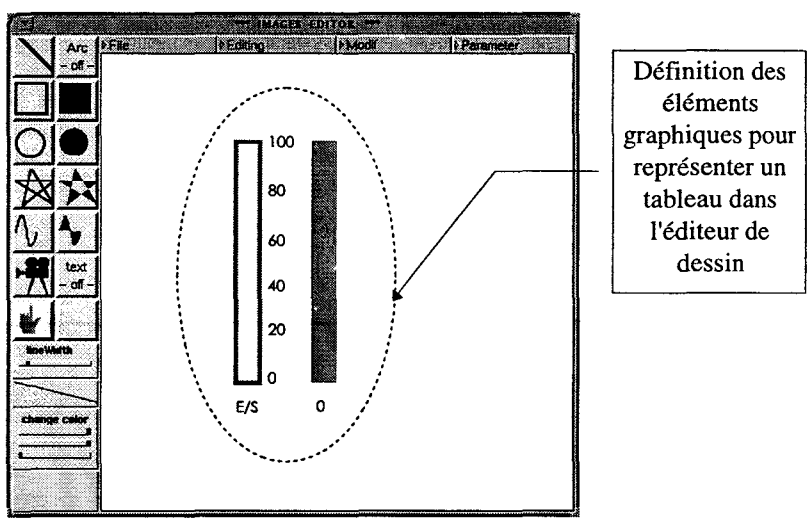


Figure 5.5 : Représentation initiale d'un tableau

Lors de la création de cette image nous ne créons qu'un seul rectangle (visualisé en grisé dans la figure ci-avant) qui nous servira de modèle de présentation et d'animation en lui adjoignant les différentes méthodes d'animation qui le concernent.

La première méthode contrôle le changement de hauteur de l'objet graphique décrit précédemment et la seconde est une copie du rectangle modèle conjuguée à un déplacement qui permet de le mettre à sa position effective dans l'image globale.

Si nous nous reportons au chapitre III.5.1, nous pouvons constater qu'il est nécessaire d'adjoindre à notre rectangle deux autres méthodes d'animation qui se traduisent d'une part, par un changement de couleur et d'autre part par une méthode de copie et déplacement différente de celle que nous venons de décrire. Celles-ci permettent de visualiser respectivement les deux derniers éléments du tableau indexé par le programme, soit lors d'une affectation ou soit lors d'une simple consultation de leur valeur et de visualiser les échanges de données.

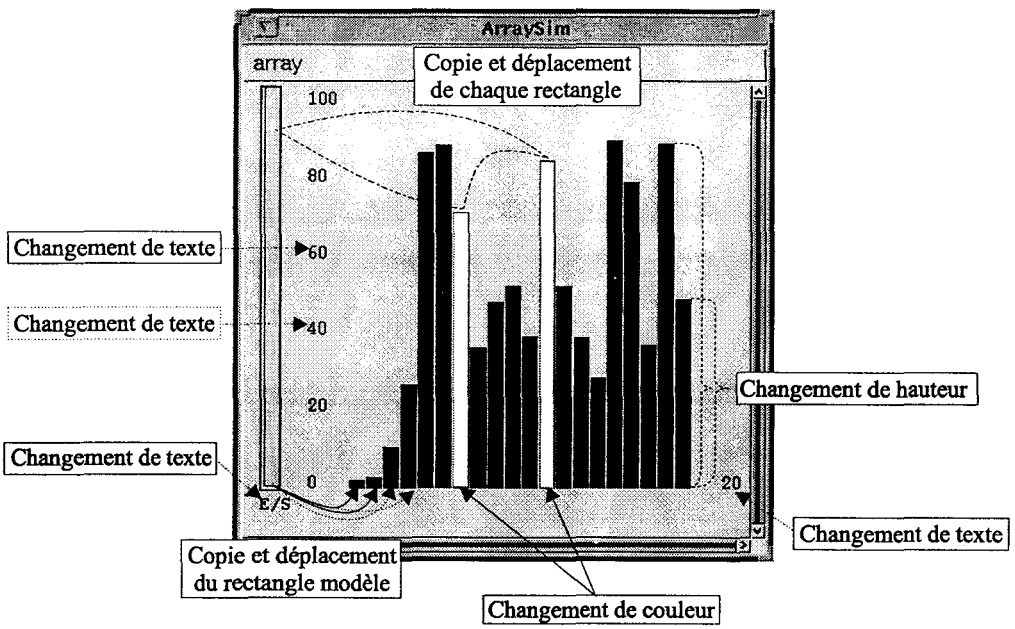


Figure 5.6 : *Animations d'une donnée de type tableau.*

Enfin, à chaque représentation textuelle nous avons adjoint une méthode de traduction changeant leur représentation visuelle ce qui nous permet alors d'afficher : l'échelle de grandeur des éléments, le nom de la variable d'échange et enfin le nombre d'éléments du tableau.

V.2.2.1.1 - Présentation et animation d'une donnée de type booléenne

D'autres visualisations de données ont été conçues en associant une image plus concrète, par exemple, à l'objet booléen nous avons joint un modèle P.Os.T. dont l'aspect visuel correspond à un feu bicolore (Cf. figure 5.7).

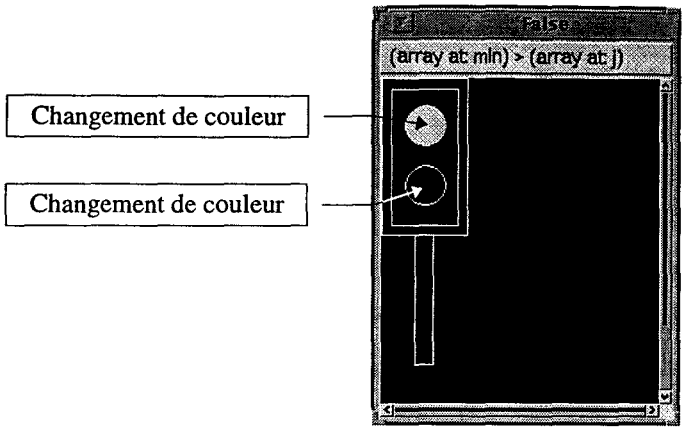


Figure 5.7 : *Visualisation d'une expression renvoyant une variable booléenne.*

Une des visualisations de ce type de donnée dans l'algorithme traité peut être obtenue en sélectionnant simplement l'expression de comparaison "(array **at:** min) > (array **at:** j)" dans la fenêtre visualisant le code du programme et en faisant appel à une des fonctions du menu fugitif associé.

L'animation de ce feu se traduit sous la forme d'un changement de couleur des deux cercles qui représentent les signaux visuels de celui-ci et correspondent aux deux états de l'objet booléen :

- ◊ "true" -> noir, "false" -> rouge, pour le premier cercle,
- ◊ "true" -> vert, "false" -> noir, pour le second.

La couleur noire a été choisie, comme nous l'avons spécifié en introduction, pour confondre l'objet graphique avec le fond et le rendre invisible.

V.2.2.2 - La visualisation de synthèse associée

Si le problème de généricité est important pour une donnée, il l'est aussi pour une visualisation de synthèse. Cependant, il ne faut pas heurter les utilisateurs dans le choix de ces représentations et conserver une certaine homogénéité et constance. Aussi, les représentations des éléments du tableau dans l'animation de synthèse sont dessinées, comme pour la donnée, sous la forme de rectangles pleins.

Nous avons de même adopté la même image graphique initiale, la différentiation ne se faisant que sur certaines méthodes d'animation associées au traducteur et dans les moyens d'obtenir les informations pour les exécuter. Si pour une donnée toutes les informations pour

l'animer lui son intrinsèques, nous pouvons pour une visualisation de synthèse faire appel à toutes les données du programme ou toutes les expressions significatives.

Celles employées pour animer la présentation de synthèse de l'algorithme de tri par sélection sont les suivantes :

- ◊ "array" ou " tab" permet de créer, de déplacer les rectangles et de vérifier la cohérence des animations,
- ◊ "temp" nous permet de contrôler la valeur d'échange et,
- ◊ "i", "j" et "min" nous permettent de changer la couleur des rectangles qui représentent les éléments que ces indices pointent (Cf. figure 5.8).

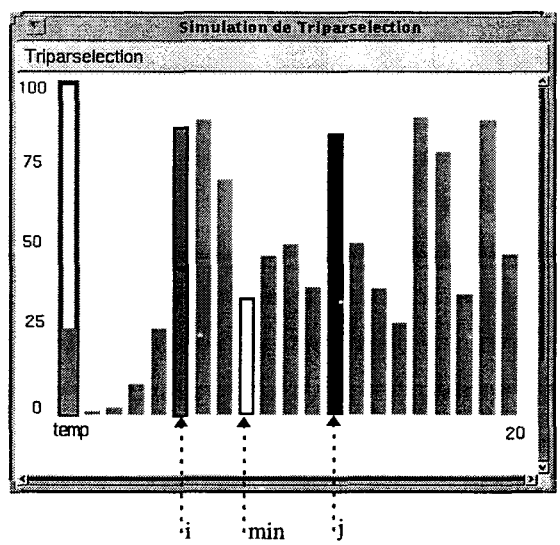


***Figure 5.8** : Visualisation de synthèse de l'algorithme de tri par sélection*

Comparativement à l'animation de la donnée tableau nous avons introduit un changement de couleur supplémentaire sur les rectangles et une gestion plus rigoureuse de l'animation en permettant de contrôler et de gérer celle-ci non plus par une capture de message mais par l'intermédiaire de l'état des variables associées. Nous obtenons ainsi une animation plus représentative car nous n'avons plus simplement une indication sur les deux derniers éléments consultés mais bien une visualisation globale des données qui interagissent avec le tableau.

L'animation de synthèse ainsi créée permet de visualiser la borne inférieure non encore triée, caractérisée par l'état de la variable "i" et qui servira d'élément d'échange, de visualiser l'élément minimum actuellement trouvé caractérisé par la variable "min" et de visualiser la recherche de ce minimum par l'intermédiaire de la variable "j". Enfin, nous pouvons visualiser l'échange entre l'élément de plus petite valeur et l'élément à la $i^{ème}$ position

par l'intermédiaire de la variable "temp" et contrôler celui-ci par l'intermédiaire de l'état du tableau et des variables "i" et "j".

V.2.3 - Conclusion

A travers cet exemple nous pouvons tirer un premier ensemble d'enseignements.

La visualisation des informations d'un algorithme dans des fenêtres autonomes permet véritablement à l'utilisateur d'établir à son gré un scénario original et montre une réel flexibilité de l'application. En outre toutes les informations sont présentées dans des fenêtres de même type et ne peuvent s'obtenir que par l'intermédiaire de la fenêtre d'exécution ce qui homogénéise l'application car elle systématise cette tâche. Cependant même si pratiquement l'utilisateur peut visualiser autant d'information qu'il le souhaite celui-ci est gêné par la taille et la résolution de l'écran ce qui limite le nombre d'information réellement présentable.

La visualisation et l'animation de la donnée tableau est pourvue d'énormément d'information sur la manière que l'algorithme a d'effectuer son tri. Cependant, pour un utilisateur naïf, cette représentation n'est pas suffisante à elle seule pour comprendre le fonctionnement global du programme. Et même si nous visualisons en même temps toutes les données qui interagissent au sein de l'algorithme il est difficile d'en faire la synthèse car elles représentent des comportements distincts. Aussi il est très souvent souhaitable de créer une visualisation de synthèse et donc d'en faciliter son élaboration car il ne peut exister de bibliothèque d'animation suffisamment riche pour représenter le comportement de tous les algorithmes que l'homme imagine. D'où l'importance des outils de prototypage graphique introduit dans le système qui permettent de construire une représentation et une animation de manière conviviale et rapide. Mais, si ces outils sont bien adaptés, le couplage de l'objet P.Os.T. à un algorithme doit encore être systématisé et facilité par le développement d'un outil spécifique.

Si lors de la création d'une représentation le souci d'homogénéité est important, la généricité l'est également et est très difficile à mettre en œuvre. La généricité demande en fait plus de temps de réflexion que d'élaboration proprement dit. Ainsi, si la représentation et l'animation de la donnée de type booléenne ont été réalisées et testées en à peine une heure, un peu plus d'une journée a été nécessaire pour réaliser celles de la donnée du type tableau. En pratique, l'expérience et la connaissance du système joue un très grand rôle dans l'élaboration d'une animation générique qui fait appel à des méthodes qui ne gèrent plus seulement des éléments graphiques mais les créent et les positionnent convenablement.

Les différentes interactions exposées au chapitre III.14 ont fait l'objet d'une validation systématique qui ne sont pas comme pour l'exemple suivant décrites car elles font appel à un cadre purement expérimental.

De même, apportant peu d'intérêt littéral, l'indépendance du code de ses animations n'est pas présentée mais a été validée en écrivant le même algorithme en Pascal auquel nous avons associé exactement la même visualisation de synthèse ce qui n'a présenté aucune difficulté.

V.3 - LE CIRCUIT HAMILTONIEN D'UN GRAPHE

Le deuxième exemple que nous allons présenter est un programme sur les graphes dont le modèle de représentation de données peut résoudre bon nombre de problèmes. Par exemple, pour trouver des parcours optimaux (pour les circuits électriques), des composants connexes (pour la classification de donnée), etc.

L'algorithme le plus représentatif se rapportant à un graphe est le cas d'un voyageur de commerce qui doit se rendre dans plusieurs villes et souhaite trouver un parcours avantageux, qui lui permette de passer dans toutes les villes en partant d'une ville initiale. La résolution de ce problème se fait par l'utilisation d'un graphe dans lequel les villes sont les noeuds et les routes les chemins reliant ces différents noeuds.

On appelle circuit hamiltonien d'un graphe un chemin qui utilise les arrêtes de ce graphe en passant une fois, et une seule, par chaque sommet avant de revenir au point de départ.

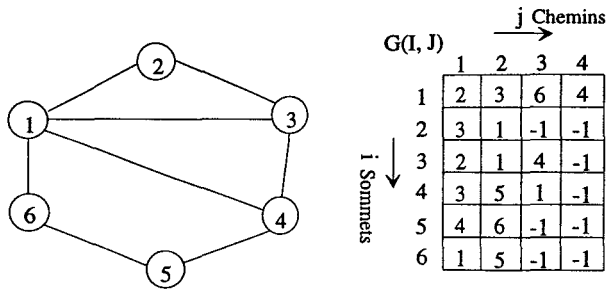
Le problème du voyageur de commerce est donc de trouver le circuit hamiltonien le plus court. Pour ce faire, nous proposons d'animer un algorithme qui recherche pour sa part tous les circuits hamiltoniens d'un graphe [SED 83].

V.3.1 - Une solution algorithmique et programmatique

Un graphe de N sommets peut être représenté par une matrice de N lignes et M colonnes, où M est le nombre maximum de chemins ou côtés partant d'un sommet.

Dans cette matrice les éléments $G(I, J)$ donneront l'ensemble des sommets Kj auquel est relié le sommet i. S'il y a moins de M liaisons, la valeur correspondante pourra, par exemple, être initialisée à -1.

Une représentation graphique et informatique d'un graphe de six sommets ayant au maximum quatre chemins est proposée en figure 5.8.



G(I, J)

j Chemins →

i Sommets ↓	1	2	3	4
1	2	3	6	4
2	3	1	-1	-1
3	2	1	4	-1
4	3	5	1	-1
5	4	6	-1	-1
6	1	5	-1	-1

Figure 5.9 : *Représentation d'un graphe et de sa matrice G associée.*

Le principe de l'algorithme consiste à partir du premier sommet et de considérer les côtés qui en partent en les essayant successivement. On avance alors dans le graphe en mémorisant les branches et les sommets déjà utilisés. Pour ce faire, l'élément du tableau *CHEMIN (I, J)*, initialisé avec les éléments *G(I, J)*, est mis à zéro et le tableau *OCCUPE (I)* prend la valeur *VRAI*.

N avances signale la découverte d'un circuit hamiltonien. Faute de quoi il faut reculer d'une étape et essayer un autre chemin.

Lorsqu'on a reculé jusqu'au sommet de départ et que toutes les branches ont été épuisées c'est qu'il n'y a plus de circuit hamiltonien.

Nous avons, dans ce programme écrit en Pascal, deux procédures appelées par le programme principal : *LIREGRAPHE* qui effectue la lecture du graphe en initialisant le tableau *"graphe" et la* procédure *HAMILTON* qui recherche les circuits hamiltoniens et qui fait appel aux procédures suivantes :

- ◊ *INITIALISATION* prépare les tableaux *CHEMIN* et *OCCUPE* ;
- ◊ *ECRIRE* permet d'écrire une solution lorsqu'elle est trouvée ;
- ◊ *AVANCE* permet d'avancer dans le graphe ;
- ◊ *RECULE* permet de reculer dans le graphe (Cf. figure 5.10).

```
program circuithamiltonien;

const n= 10;
      m=10;

type reseau = array[1..n,1..m] of integer;

var nn, mm, nbSol, i, j : integer;
    graphe, chemin : reseau;
    occupe: array[1..n] of boolean;
    solution: array [0..n] of integer;
```

```
procedure liregraphe(var graphe: reseau; n, m: integer);
var i, j: integer;
begin
    for i:=1 to n do
        begin
            writeln('Entrer le noeud n— ',i);
            j := 1;
            repeat
                read(graphe[i,j]);
                write(' ');
                j := succ(j);
            until j > m;
            writeln;
        end;
end;
```

```
procedure initialisation(n, m: integer);
var i, j :integer;
begin
    for i := 1 to n do
        begin
            occupe[i] := false;
            for j := 1 to m do
                chemin[i,j] := graphe[i,j];
        end;
end;


procedure avance(var sommet, i, j: integer);
begin
    solution[sommet+1] := chemin [i,j];
    chemin [i,j] := 0;
    i := solution[succ(sommet)];
    occupe[i] := true;
    sommet := succ(sommet);
end;


procedure recule(var sommet, i: integer);
var j : integer;

begin
    for j := 1 to mm do
        chemin [i,j] := graphe [i,j];
    occupe[i] := false;
    sommet := pred(sommet);
    i := solution[sommet];
end;


procedure ecrire(var nbsol: integer);
var k : integer;

begin
    writeln;
    writeln('solution n° ', nbsol);
    for k := 1 to nn do
        write('  ',solution[k]);
    nbsol := nbsol + 1;
    writeln;
end;
```

```
procedure hamilton(graphe: reseau; n, m : integer);
var i, j , sommet : integer;
    avant: boolean;

begin
    initialisation(n,m);
    sommet := 1;
    nbsol := 1;
    solution[1] := 1;
    occupe[sommet] := true;
    i := 1;
    while sommet > 0 do
        begin
            j := 1;
            avant := false;
            while j <= m do
                if chemin [i,j] <> 0 then
                    if chemin [i,j] > 0 then
                        if not occupe[chemin[i,j]] then
                            begin
                                avance(sommet, i, j);
                                if sommet = n then
                                        ecrire(nbsol)
                                    else avant := true;
                                j := m + 1;
                            end
                            else j := j + 1
                        else j := m + 1
                    else j := j + 1;
            if not avant then recule(sommet, i);
        end;
end;


begin
    writeln('entrer le nombre de sommets');
    readln(nn);
    writeln('nombre maximum de cotes partant d''un
            sommet');
    readln(mm);
    liregraphe(graphe, nn, mm);
    writeln('circuits hamiltoniens');
    hamilton(graphe, nn, mm);
end.
```

Figure 5.10 : *Un programme Pascal implémentant l'algorithme de recherche des circuits hamiltoniens d'un graphe.*

V.3.2 - Présentation et animation d'une donnée de type tableau à deux dimensions

Comme nous l'avons vu précédemment pour visualiser une donnée de type "graphe" ou "chemin" du programme ci-dessus un modèle P.Os.T. doit lui être associé. Concrètement, en Smalltalk une matrice est un tableau de tableau et se réfère donc à la classe "Array". Cet

objet ne possède alors, non pas un, mais deux modèles de représentation choisis par la méthode d'affichage selon que l'objet receveur a une ou deux dimensions.

Pour une matrice de, M lignes, N colonnes, l'animation s'effectue en gérant M + 1 lignes et N + 1 colonnes équidistantes dont l'une à pour longueur, M * sa longueur initiale, et l'autre pour hauteur N * sa hauteur initiale. Ceci est créé en donnant aux deux éléments graphiques servant de modèle la méthode de copie et de déplacement vue précédemment pour le tableau à une dimension ainsi que la méthode de changement de hauteur pour l'un et une méthode de changement de longueur pour l'autre.

D'autre part, les éléments de la matrice apparaissent sous forme textuelle et sont positionnés au centre du rectangle auquel ils se réfèrent. Aussi, une nouvelle méthode de copie et déplacement d'un élément graphique est insérée au sein du traducteur et fait référence aux différents objets textuels. De même, une méthode de changement de texte leur a été attribuée permettant d'afficher la valeur des éléments qu'ils représentent (Cf. figure 5.11).

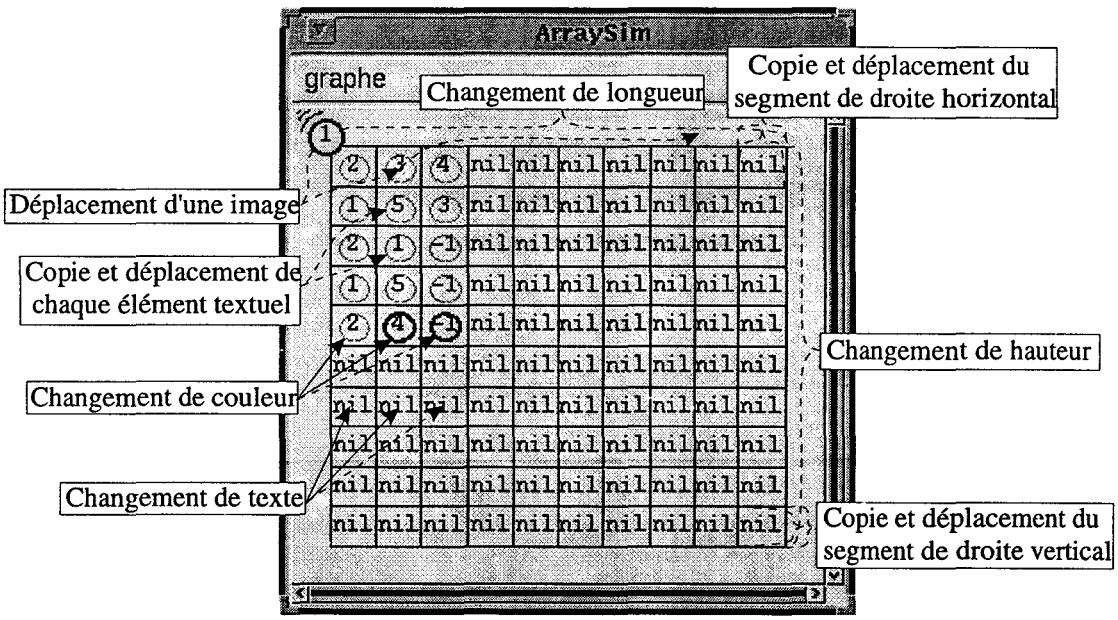


Figure 5.11 : *Visualisation de la donnée "graphe"*

Comme dans un tableau à deux dimensions, nous visualisons les deux derniers éléments consultés en les entourant d'un cercle vide dont l'animation est un simple changement de couleur. Et enfin nous avons associé une méthode de déplacement d'image composée d'un cercle plein et d'une valeur textuelle pour visualiser les différentes affectations lors d'échange de donnée de la matrice avec une donnée externe.

V.3.3 - La visualisation de synthèse associée

Si nous nous reportons à la définition d'un graphe, nous constatons que celui-ci est composé de noeuds et de chemins décrivant les relations entre ceux-ci (cf. figure 5.9).

Pour obtenir une visualisation de synthèse générique, s'il est possible de créer des noeuds en se basant sur un élément graphique servant de modèle et de les placer en leur affectant un ensemble de positions prédéfinies, ce n'est pas le cas pour les chemins qui doivent être créés sans engendrer d'erreur d'interprétation relationnelle (Cf. figure 5.12).

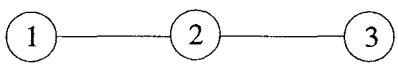


Figure 5.12 : *Exemple de relation entre les noeuds "1" et "3" qui peut être confondue en une double relation entre les noeuds "1", "2" et "2", "3"*

Nous avons donc adopté une image initiale d'un graphe ayant 10 noeuds visualisables sous forme de rectangles pleins, d'un texte d'identification et d'un rectangle vide de sélection. Ce nombre est symbolique mais correspond comme nous pouvons le voir en figure 5.13 à une limite proche de la réalité. Tous les chemins peuvent être alors préétablis et apparaître sous forme de lignes continues et discontinues auxquelles nous avons associé un texte qui permet de visualiser un attribut de longueur ou tout autre paramètre.

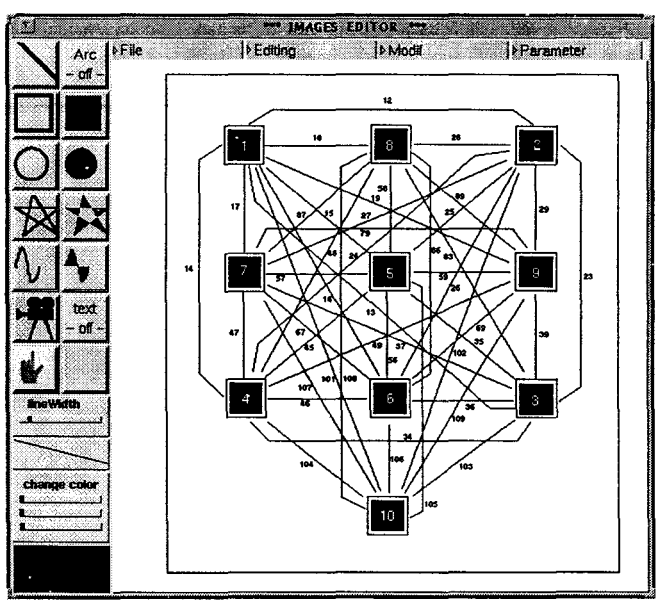


Figure 5.13 : *Représentation initiale d'un graphe de 10 noeuds*

Pour rendre partiellement générique cette animation, un moyen adopté est de mettre initialement tous les éléments graphiques dans la couleur de fond de l'image, ce qui les rend alors visuellement inexistants. Puis en associant des méthodes d'animation de changement de couleur à chacun de ces éléments nous pouvons présenter le graphe initialisé par l'utilisateur ainsi que les interactions du programme avec celui-ci.

Nous avons également ajouté aux différents textes une animation permettant de changer leur représentation visuelle car d'une part, les noeuds peuvent être référencés par d'autres objets que des numéros et d'autre part, les paramètres attachés aux différents chemins sont des variables de programmes.

A chaque donnée de l'algorithme de recherche des circuits hamiltoniens a donc été associée une méthode d'animation de changement de couleur. La variable "nn" définissant le nombre de sommets du graphe fait donc apparaître "nn" rectangles. L'initialisation de la variable "graphe", quant à elle, fait apparaître successivement les chemins définis. La variable "i" représentant le noeud actif quelle que soit la procédure exécutée est associée aux différents rectangles de sélection. L'affectation de la variable "occupe" fait également changer de couleur selon sa valeur le noeud affecté (false -> rouge, true -> vert) faisant apparaître successivement en vert les noeuds validés (visualisé en grisé pâle dans la figure 5.14). L'association des variables "i" et "j" nous permet de visualiser le parcours actuellement testé par le programme en changeant de couleur le chemin que ces deux données spécifient. Enfin, la variable "chemin" nous permet de visualiser le parcours validé ou fini d'être testé en changeant de couleur les chemins mémorisés.

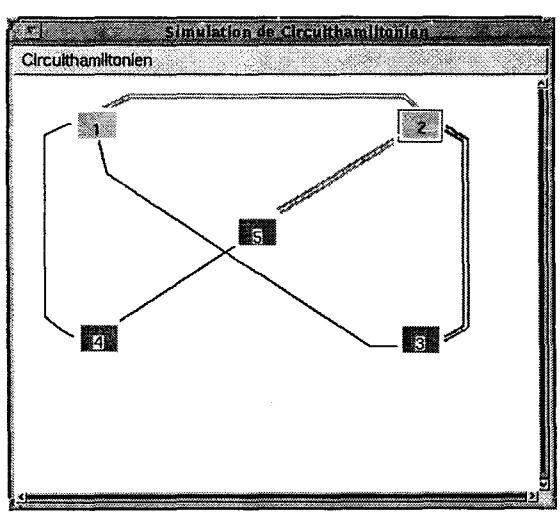


Figure 5.14 : *Visualisation de synthèse de l'algorithme de recherche des circuits hamiltoniens d'un graphe*

L'animation ainsi définie nous permet de rendre compte de l'activité du programme et de visualiser sa méthodologie de recherche.

Par exemple, dans la visualisation de synthèse présentée en figure 5.13 nous pouvons voir que nous avons défini un graphe de cinq noeuds comportant pour chacun au maximum trois chemins. Le programme est passé, lors de sa recherche, successivement du noeud "1" au noeud "2" et actuellement nous sommes en train de tester les différents chemins qui lui sont associés. Ainsi, le programme a essayé de passer, sans succès, par le noeud "5" et essaye maintenant de passer par le noeud "3".

V.3.4 - Conclusion

Pour que la visualisation d'un tableau à deux dimensions soit générique l'utilisateur est amené à utiliser des méthodes d'animations complexes. Cependant si elles sont appliquées à des éléments graphiques différents, ces méthodes sont les mêmes que celles employées pour visualiser les éléments d'un tableau à une dimension (cf. paragraphe V.2.2.1.1). Aussi l'expérience du système est véritablement l'élément déterminant pour élaborer une animation complexe. Néanmoins quel que soit le niveau de connaissance de l'utilisateur du système une représentation de synthèse générique n'est pas toujours réalisable.

Par exemple, les difficultés rencontrées pour visualiser correctement et distinctement les chemins d'un graphe nous ont conduit à associer à l'algorithme une visualisation de synthèse préétablie qui limite le nombre maximum de noeud définissable au chiffre symbolique de dix. Cependant l'avantage de ce type de représentation est qu'elle fait appel à des méthodes d'animation plus conventionnelles ce qui facilite du même coup son élaboration. Aussi le fait de pouvoir réaliser une représentation de synthèse simple mais spécifique permet au système de s'adapter à l'expérience de l'utilisateur.

Pour cet exemple, la mise en oeuvre d'une animation générique doit résoudre un véritable problème de contraintes graphiques telle qu'elle a été exposée au chapitre II.5.4.2.

V.4 - FILTRAGE NUMERIQUE D'UNE IMAGE

L'algorithme que nous allons maintenant étudier concerne le domaine du traitement d'images et plus concrètement les différents pré-traitements que l'on peut effectuer en vue d'une reconnaissance de forme ou d'une segmentation [HOR 93].

En fait, cette étape de pré-traitement a pour objet de renforcer la ressemblance entre pixels pour partitionner l'image en régions homogènes.

Ce type d'algorithme a pour caractéristique d'effectuer un certain nombre d'opérations de filtrage sur une image source constituant différents processus élémentaires dont la

décomposition est particulièrement bien adaptée à une programmation visuelle sous forme modulaire.

V.4.1 - Quelques algorithmes de pré-traitement

Le moyen le plus classique pour extraire les contours des régions d'une image est d'effectuer un rehaussement du contraste par l'intermédiaire d'une opération de filtrage spatial passe-haut (différenciation) dont les plus simples utilisent les dérivées partielles de la fonction niveau de gris.

Le contour correspond à des changements brusques des propriétés physiques des objets dans l'image, comme la discontinuité, qui correspondent à des variations locales de niveaux de gris importantes, détectables par la dérivée première ou seconde de la fonction image. Ces dérivées peuvent être ramenées à une convolution de l'image par une fenêtre de dimensions finies, c'est-à-dire pour laquelle chaque point de l'image traitée est fonction, non seulement du point correspondant de l'image d'entrée mais de tous les points voisins contenus dans la fenêtre.

Ainsi plusieurs opérateurs dérivatifs ont été trouvés et peuvent être mise en oeuvre dans notre application (opérateurs de Prewitt, de Sobel, de Kirsh, du laplacien, etc.).

Ayant rehaussé le contraste d'une image, l'opération suivante consiste à déterminer les régions d'intérêt dans la scène étudiée et d'éliminer les transitions de faible gradient par un opérateur de seuillage. Le plus élémentaire est une binarisation d'une image source à plusieurs niveaux de gris qui consiste à obtenir une image résultat dont tous les pixels, de valeurs inférieures à un seuil deviennent noirs et tous ceux supérieurs deviennent blancs. Pour déterminer ce seuil nous avons utilisé la méthode la plus classique qui consiste préalablement à visualiser l'histogramme de l'image construit sur la fréquence d'apparition dans l'image de chaque niveau de gris et de déterminer à partir de celui-ci le ou les minimums locaux à travers ces modes.

V.4.2 - Implantation d'un algorithme de pré-traitement

Ayant choisi d'implanter l'algorithme de pré-traitement d'une image par l'intermédiaire de l'outil de programmation visuelle, nous avons associé à toutes ces tâches un algorithme et donc un module particulier. D'autre part, une image et animation particulière peuvent être associées à la présentation de ces modules conformément au modèle P.Os.T. (cf. chapitre IV.4.2).

V.4.2.1 - Implantation et animation d'un module masque

Tous les filtres susceptibles d'agrémenter cet exemple appliquent un masque de calcul de huit voisins sur un pixel donnée et translate celui-ci sur toute l'image source.

Par exemple, les opérateurs de Prewitt et de Sobel font appel à deux masques qui détectent les dérivées directionnelles horizontale et verticale et s'expriment sous la forme :

$$h_i = \begin{bmatrix} 1 & 0 & -1 \\ c & 0 & -c \\ 1 & 0 & -1 \end{bmatrix} \qquad h_j = \begin{bmatrix} 1 & c & 1 \\ 0 & 0 & 0 \\ -1 & -c & -1 \end{bmatrix} \qquad \text{avec:} \quad \begin{matrix} c = 1, \text{ pour Prewitt} \\ c = 2, \text{ pour Sobel} \end{matrix} \qquad \text{et}$$

pour le pixel référencé par les index i et j, si E est l'image source ; X et Y le résultat de l'application du masque de calcul de huit voisins sur E s'expriment sous la forme :

$$X_{(i,\,j)} = \frac{1}{4+2c} h_i * \begin{bmatrix} E_{(i-1,j-1)} & E_{(i,j-1)} & E_{(i+1,j-1)} \\ E_{(i-1,j)} & E_{(i,j)} & E_{(i+1,j)} \\ E_{(i-1,j+1)} & E_{(i,j+1)} & E_{(i+1,j+1)} \end{bmatrix} \quad \text{et,}$$

$$Y_{(i,\,j)} = \frac{1}{4+2c} h_j * \begin{bmatrix} E_{(i-1,j-1)} & E_{(i,j-1)} & E_{(i+1,j-1)} \\ E_{(i-1,j)} & E_{(i,j)} & E_{(i+1,j)} \\ E_{(i-1,j+1)} & E_{(i,j+1)} & E_{(i+1,j+1)} \end{bmatrix}$$

enfin la norme du gradient S donnant l'image finale après l'application des deux masques est donnée par :

$$S_{(i,\,j)} = \left[X^2_{(i,\,j)} + Y^2_{(i,\,j)} \right]^{1/2}$$

Nous avons donc, tout d'abord, créé un module qui applique un masque quelconque sur une image dont la fenêtre correspond à 3*3 pixels dont le code d'implémentation se trouve ci-après.

```
operat9Masque: image
    | size newImage num array |
    newImage := image copy.
    size := newImage extent.
    array := #(1 1 1 0 0 0 -1 -1 -1 ).
    num := 0.
    1 to: 9 do: [:i | num := num + (array at: i) abs ].
    2 to: size x - 1
      do: [:x | 2 to: size y - 1
              do: [:y | newImage atX: x
                                y: y
                                put: ((array at: 1) * (image atPoint: x - 1 @ (y - 1)) +
                                     ((array at: 2) * (image atPoint: x @ (y - 1))) +
                                     ((array at: 3) * (image atPoint: x + 1 @ (y - 1))) +
                                     ((array at: 4) * (image atPoint: x - 1 @ y)) +
                                     ((array at: 5) * (image atPoint: x @ y)) +
                                     ((array at: 6) * (image atPoint: x + 1 @ y)) +
                                     ((array at: 7) * (image atPoint: x - 1 @ (y + 1))) +
                                     ((array at: 8) * (image atPoint: x @ (y + 1))) +
                                     ((array at: 9) * (image atPoint: x + 1 @ (y + 1))) / num) rounded]].
    ^newImage
```

" [1 1 1]
[0 0 0]
[-1 -1 -1]"

Pour obtenir le filtre effectivement désiré il faut modifier la partie de ce code après avoir introduit le module dans la plate forme de programmation visuelle

Cet algorithme n'a pas été construit de manière optimale car celui-ci correspond à un masque particulier. Cependant, si la donnée "array" du module servant de modèle a été initialisée avec différentes valeurs, l'utilisateur peut altérer les différents codes des composants logiciels qu'il emploie indépendamment les uns des autres pour obtenir l'opérateur désiré grâce aux capacités d'interaction de notre environnement. Pour ce faire, avant d'exécuter le programme visuellement créé, l'utilisateur sélectionne les modules qu'il désire modifier et fait appel au menu contextuel associé à chacun de ceux-ci pour ouvrir une fenêtre d'édition permettant de visualiser leur code (cf. chapitre IV.7). Nous validons ainsi certaines facultés de notre système à rendre générique et polymorphe un module, à partir d'un algorithme de base qui ne l'est pas.

Le choix de la représentation du module est comme nous pouvons le voir en figure 5.15 assez simple.

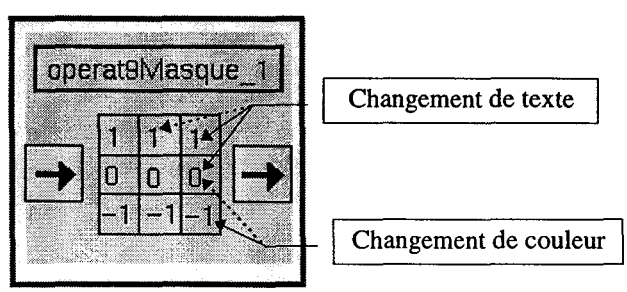


Figure 5.15 : *Visualisation du module "masque" dans l'outil de programmation visuelle*

L'animation associée suffit à présenter d'une part le masque effectivement utilisé qui change le texte des éléments de la matrice visualisée et d'autre part l'activité du module en

modifiant la couleur de l'élément évalué. Pour ce faire, nous employons les mêmes concepts de transmission d'informations vues pour l'animation de synthèse d'un algorithme (cf. chapitre III.6) qui consiste à adjoindre à la présentation du module la ou les variables intéressantes du programme (par exemple, la variable "array" pour le module ci-dessus).

V.4.2.2 - Présentation d'un module histogramme

Une présentation et une animation spécifiques ont été également réalisées pour visualiser l'histogramme d'une image et sont comparables à la visualisation d'un tableau (Cf. figure 5.16).

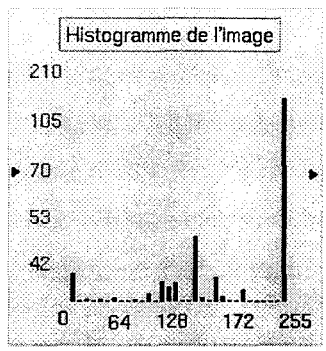


***Figure 5.16 :** Représentation du module histogramme*

Nous pouvons remarquer que nous n'avons pas associé un rectangle à chaque niveau de gris. En fait, nous nous sommes intentionnellement limités à ne représenter que 256 modulo 8 éléments, ce qui est la plupart du temps suffisamment représentatif. Si toutefois ce n'est pas le cas, c'est que l'image présente un histogramme restreint sur quelques couleurs et que pour la traiter il convient d'en effectuer l'expansion par un opérateur adapté.

En ce qui concerne les animations associées, nous retrouvons deux types utilisés pour le tableau : les changements de texte pour visualiser l'échelle de grandeur des éléments qui correspondent au nombre de pixels ayant un certain niveau de gris et le changement de hauteur attribuée à chaque rectangle en ayant pour contrainte d'afficher l'objet graphique proportionnellement à la plus grande des valeurs.

Ainsi lorsque l'utilisateur fera appel à l'algorithme chargé de spécifier la répartition des niveaux de gris d'une image sa visualisation dans la plate-forme de programmation visuelle aura pour forme la figure 5.15 qui est la représentation graphique et animée de son comportement.

V.4.2.3 - Visualisation de l'algorithme

L'algorithme que nous avons implémenté a pour objet de tester l'efficacité de différents filtres de pré-traitement comprenant l'opérateur de Prewitt et l'opérateur laplacien sur un voisinage réduit qui peuvent être vus comme un séquencement de tâches.

⇒ Lissage de l'image source.
⇒ Calcul de la dérivée, ou du laplacien en chaque point de l'image.
⇒ Création de l'image de la norme du gradient avec extraction des maximums locaux.
⇒ Seuillage par binarisation.

Pour ce faire nous avons construit dans notre environnement d'autres modules de traitement d'image tels que : son acquisition suivie de sa transformation en niveau de gris, sa visualisation dans une fenêtre autonome, sa binarisation, le calcul de la norme de deux images, etc., auxquels nous avons éventuellement associé une représentation personnalisée (cf. tableau 5.1).

Faisant ainsi appel à de nombreux composants logiciels, nous avons encapsulé quelques modules de l'opérateur de Prewitt pour qu'ils puissent être convenablement visualisés dans la fenêtre de programmation visuelle initiale.

Nous pouvons voir (Cf. figure 5.18) que les différents masques correspondant à l'opérateur de Prewitt et laplacien (Cf. figure 5.17, page 1) ont été implémentés par l'intermédiaire du module initial précédemment décrit.

Les modules d'histogramme d'une image des deux opérateurs ont été implantés dans la fenêtre principale pour permettre de déterminer le seuil de binarisation de ces deux filtres sans être obligé de visualiser tout le traitement du programme.

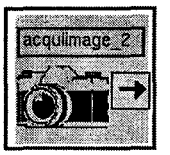
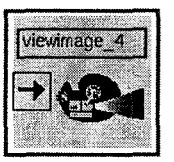
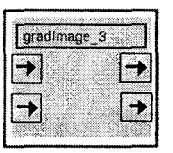
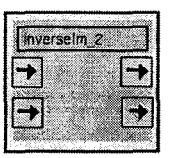
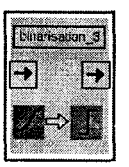
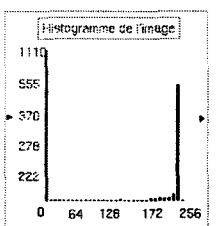
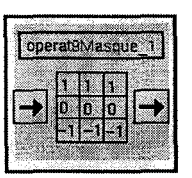
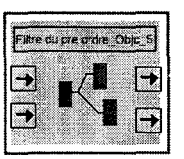
Module	**Paramètres du module**	**Présentation du module**	**Variable animée**	**Elément graphique animé**	**Type d'animation**
Acquisition d'une image	anImage		-	-	-
Visualisation d'une image	anImage		-	-	-
Gradient d'une image	firstIm, secIm, newImage		-	-	-
Inversion d'une image	image, newImage		-	-	-
Binarisation d'une image	anImage, newImage		-	-	-
Histogramme des niveaux de gris d'une image	imageEnt, imageSort		array	rectangles textes	changement de hauteur changement de texte
Masque d'une image	image, newImage		array	textes	changement de texte, changement de couleur
Encapsulation	-		-	-	-

Tableau 5.1: *Liste des modules utilisés pour implémenter l'algorithme de pré-traitement d'une image*

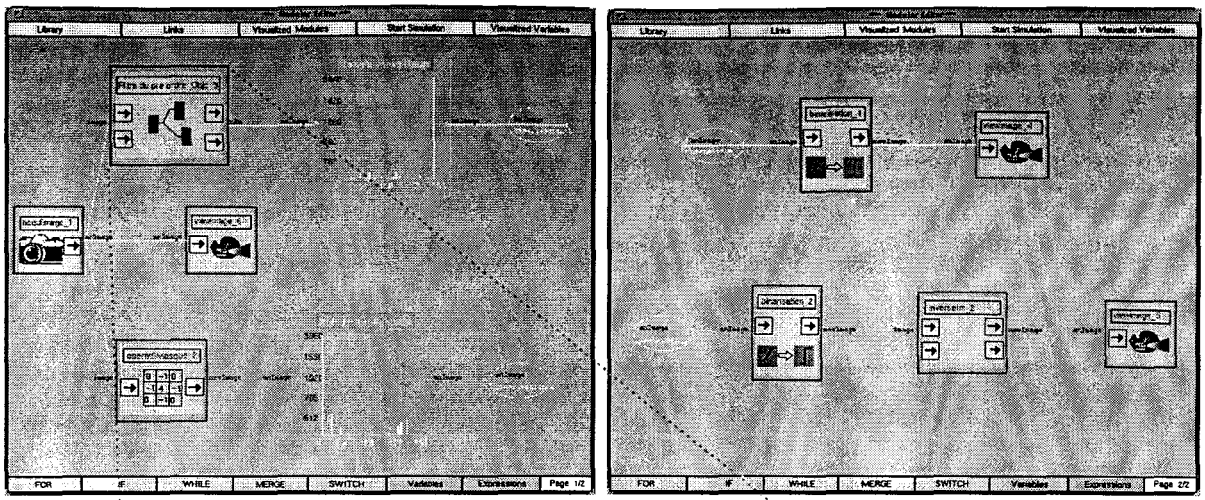


***Figure 5.17** : Fenêtre principale du programme modulaire de pré-traitement d'une image implanté sur deux pages*

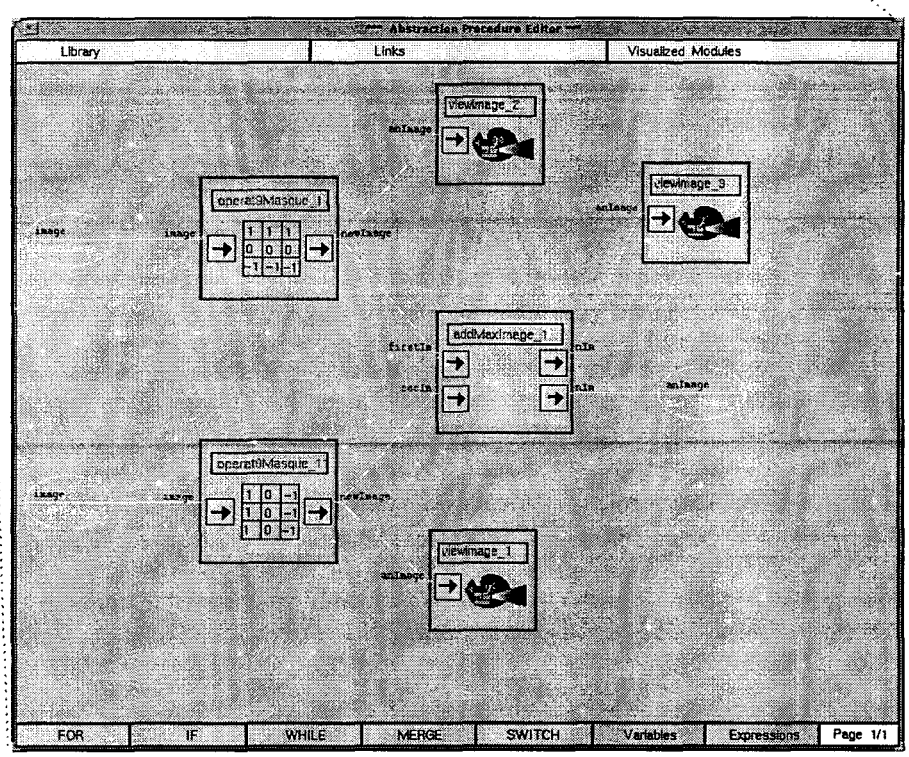
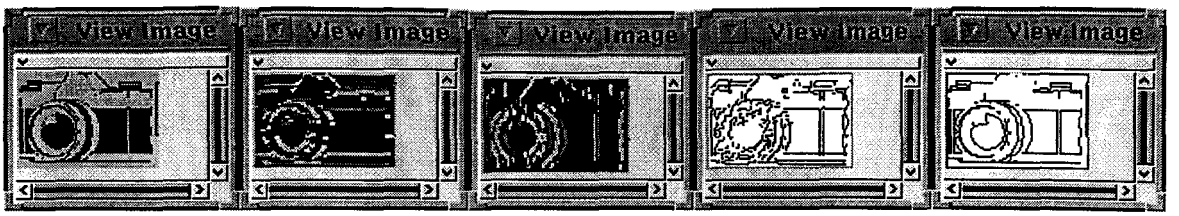


***Figure 5.18** : Fenêtre de visualisation du pré-traitement de l'opérateur de Prewitt.*

L'utilisation de l'outil de programmation modulaire a permis, en définissant un simple lien, d'associer après chaque pré-traitement le même module de visualisation d'image et ainsi de regarder les effets de ceux-ci, dans des fenêtres autonomes (Cf. figure 5.19).



(a) (b) (c) (d) (e)

Figure 5.19: *Résultats du pré-traitement de l'image originale (a) avec présentation en (b) et (c) du résultat du passage du masque horizontal et vertical de Prewitt et en (d) l'image finale après calcul du gradient et de sa norme suivi de sa binarisation. Enfin, en (e) l'image finale pour une dérivation du second ordre correspondant au Laplacien d'une image.*

En plus de l'animation apportée à certains modules pour faire transparaître son activité, l'utilisateur peut visualiser plus explicitement les informations qui ont trait à chaque module constituant le programme lors de leur évaluation, soit par l'intermédiaire de l'outil de visualisation de programme pour un ou plusieurs modules spécifiques (Cf. figure 5.20) ou soit indépendamment de celui-ci en inspectant seulement l'état des variables du programme dans des fenêtres autonomes (Cf. chapitre IV.6, figure 5.21).

Nous pouvons par exemple visualiser le traitement du masque vertical de Prewitt et ponctuellement l'effet de ce calcul sur l'image résultat ou regarder simplement l'évolution de l'état de la variable "newImage" définie dans plusieurs modules du programme.

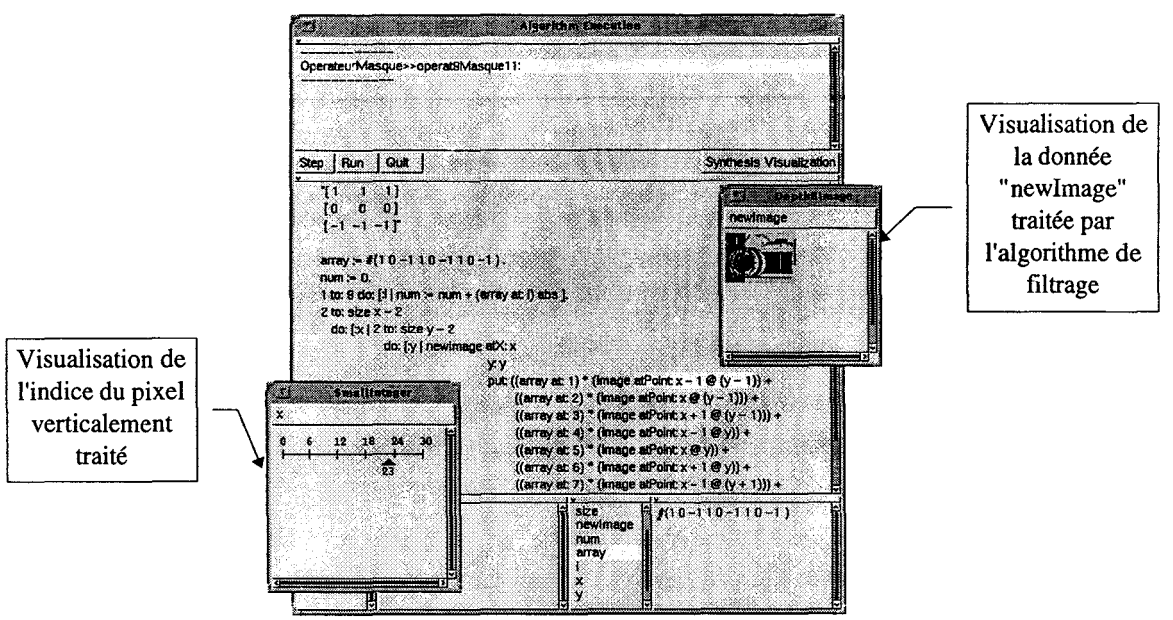


Figure 5.20 : *Visualisation du traitement d'un masque de Prewitt et des informations associées*

Figure 5.21 : *Visualisation de la donnée "newImage" au cours de l'évaluation du module déterminant la norme du gradient d'une image*

V.4.3 - Conclusion

L'outil de programmation visuelle permet de constituer rapidement des algorithmes en utilisant plusieurs modules de base identiques et de vérifier le traitement de chacun d'eux après ou lors de l'exécution de la tâche. Ainsi, toutes les visualisations d'information aident l'utilisateur à se rendre compte de quelle façon les modules agissent effectivement sur les données du programme et peuvent permettre entre autres de déterminer quel code ou quelle donnée génère une erreur, quel traitement est inefficace pour un paramètre donné, etc. Ceci n'est en fait possible que par les facilités d'interaction que nous avons développées et maintenues tout au long de l'élaboration de notre système.

En dehors de l'efficacité de l'utilisation d'un tel outil, la personnalisation graphique de la représentation des modules et leur animation permet de visualiser en un instant ce qu'ils effectuent et leurs activités. En outre l'utilisation des mêmes outils pour élaborer ces représentations ou les informations d'un algorithme élémentaire homogénéise l'environnement et permet à l'utilisateur d'accroître son expérience très rapidement. Concrètement, l'approche visuelle est identique et consiste à attribuer à un programme ou à chaque module qui le compose une représentation et une animation représentative du traitement qu'il effectue.

V.5 - ASSERVISSEMENT D'UNE ANTENNE EN POSITION ET EN POURSUITE

Le dernier exemple présenté est un algorithme qui nous permet de régler les paramètres d'asservissement d'un moteur à courant continu d'une antenne modélisée sous la forme d'une fonction de transfert.

Concrètement, l'automaticien fait de plus en plus appel à des méthodes numériques, à l'aide d'ordinateurs pour la résolution des équations différentielles. En fait, les calculs ne se déroulent pas en temps réel et les résultats sont affichés graphiquement sous forme de courbes sur une table traçante ou un écran graphique.

La simulation est un moyen très puissant pour vérifier la validité de la conception d'un système automatique en général et de la configuration et de la dimension des systèmes de réglage en particulier.

Avec la simulation numérique, on calcule l'allure temporelle de phénomènes transitoires. On peut alors analyser le comportement du système à régler, des organes de commande ou de mesure, y compris les régulateurs [LAN 93].

V.5.1 - But de l'asservissement d'une antenne

L'asservissement du moteur d'une antenne est le type même de l'asservissement dont les performances doivent être très élevées dus aux fonctions qui lui sont demandées (suivi d'une fusée, d'un satellite, etc.)

En particulier les points suivants doivent être satisfaits :

- pas d'erreur de position (mis à part celle due aux imprécisions du matériel en lui-même),
- absence de dépassement lors d'un brusque changement de position (échelon), de façon à obtenir la rapidité maximale de changement de cap,
- possibilité de suivre une consigne sans erreur de traînage (suivi d'un objectif mobile).

V.5.2 - Fonction de transfert du moteur et correction P.I.D.

Un système au sens de l'automaticien est souvent représenté par un schéma symbolique formé d'une succession de fonctions de transfert élémentaires et de liaisons reliant celles-ci. Elles sont d'autre part généralement modélisées sous la forme de blocs fonctionnels qui peuvent être assimilés à des modules autonomes ayant pour comportement une relation mathématique entre leurs paramètres d'entrée et de sortie.

Nous supposerons que l'identification du moteur a été effectuée et que sa fonction de transfert en "p" peut s'écrire comme à la figure 5.22.

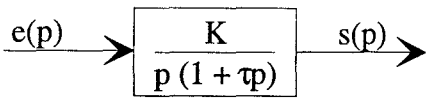


***Figure 5.22 :** Fonction de transfert du moteur visualisé sous la forme d'un bloc fonctionnel*

Pour résoudre le problème de la commande d'un moteur, plusieurs méthodes sont utilisables. Pour une grande partie d'entre elles, il s'agit de mettre au point un correcteur que l'on insère dans le système bouclé (cf. figure 5.23).

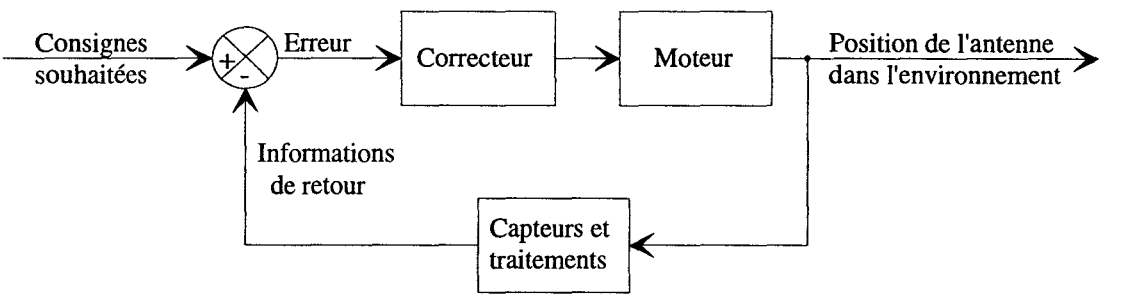


***Figure 5.23 :** Schéma fonctionnel de la commande du moteur*

Ainsi le correcteur que nous allons implémenter est une commande P.I.D. (Proportionnelle, Intégrale, dérivée) classique qui combine les avantages des régulateurs PI et PD (Cf. figure 5.24).

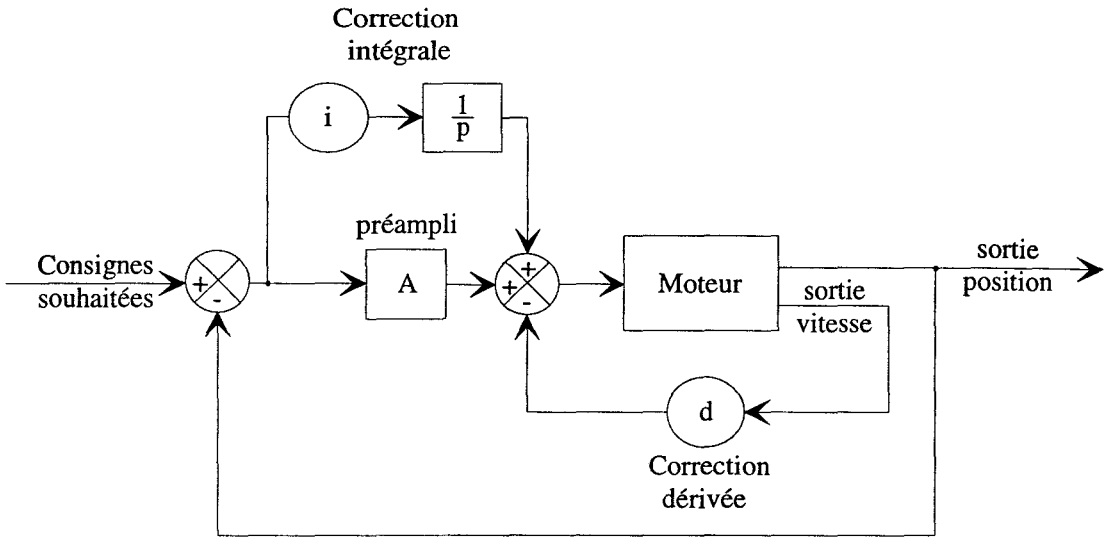


Figure 5.24 : *Schéma fonctionnel de la régulation du moteur*

Les paramètres "d" et "i" permettent respectivement de régler la correction dérivée et la correction intégrale pour différentes valeurs de gain "A". Les actions apportées par ce correcteur sont alors :

- l'action proportionnelle en statique diminue l'erreur si le gain augmente et en dynamique, la rapidité augmente tant que le système n'est pas trop oscillatoire,

- l'action intégrale en statique élimine l'erreur entre la consigne et la mesure mais, en dynamique, elle diminue la rapidité et augmente l'instabilité,

- l'action dérivée en statique n'a aucun effet ; par contre en dynamique elle augmente la rapidité grâce à son effet stabilisant.

V.5.3 - Implantation de l'algorithme

L'implantation d'un tel algorithme doit d'une part permettre à un utilisateur de décrire le schéma synoptique du système différentiel et contrôler sa simulation temporelle. De même, celui-ci doit pouvoir appliquer sur l'entrée du synoptique qu'il aura construit un certain signal et observer la sortie. La simulation temporelle fait alors appel à une discrétisation des signaux avec un pas d'échantillonnage suffisamment faible pour que la réponse soit proche de celle du système continu étudié. Le calcul se fait par une approximation des fonctions d'intégration à l'aide de méthodes classiques (Runge-Kuta).

V.5.3.1 - Modules utilisés

Comme nous venons de le voir un système qui est modélisé sous forme de blocs fonctionnels peut être décomposé en différents modules.

Ainsi, la fonction de transfert du moteur correspond à un algorithme particulier qui peut être utilisé comme un composant logiciel dans la plate-forme de programmation visuelle. Cette fonction ainsi que les boucles d'asservissements pouvant être paramétrées ; nous faisons appel à des déclarations de variables sous forme de composants logiciels (cf. chapitre IV.4.6). L'utilisateur peut à loisir changer le gain "K" du moteur, sa constante de temps "τ", le gain "A" de l'asservissement, les paramètres "d" et "i" et visualiser l'effet de chacun d'eux.

De la même façon, le signal d'entrée correspondant à une fonction de transfert peut également être modélisé sous la forme d'un bloc fonctionnel et ainsi correspondre à un module autonome.

La difficulté de cette implantation réside dans la gestion du temps de simulation qui est défini entre deux bornes correspondant à un instant initial t_0 et une valeur finale t_f, choisis selon le processus simulé pour en observer le régime transitoire ou le régime permanent. Cet intervalle de temps est alors divisé en un nombre de pas de calcul qui doit pouvoir être fixé par l'utilisateur pour obtenir une simulation correcte du système.

Pour résoudre ce problème nous conjuguons l'effet de deux modules correspondant à un "compteur-décompteur" et à une boucle "for" qui permettent respectivement de contrôler le temps entre deux échantillons et l'intervalle de temps avec $t_0 = 0$.

Le module "compteur-décompteur" compte ou décompte selon un intervalle que nous avons appelé "pas" et qui peut-être défini suivant une valeur définie par un module externe.

Chacun de ces modules implantés dans la plate-forme de programmation visuelle fait appel à un modèle de présentation et d'animation P.Os.T. personnalisé dont les animations associées correspondent principalement à des changements de texte (cf. tableau 5.2).

Pour visualiser le tracé de la réponse du système nous faisons appel à un module particulier qui nous permet de visualiser l'historique de deux données en mémorisant successivement leurs valeurs.

La représentation graphique de ce module est comparable à un écran d'oscilloscope dont les signaux sont visualisés sous la forme de points graphiques (cf. figure 5.25).

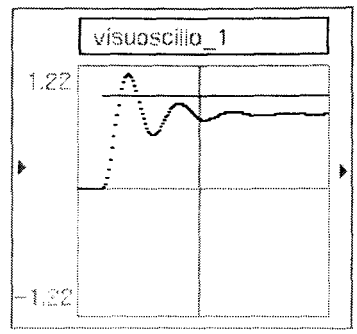


Figure 5.25 : *Représentation du module permettant de visualiser l'historique de deux données*

Pour que chacun de ces points corresponde effectivement aux valeurs successives des données d'entrées, ceux-ci sont enregistrés par le module sous la forme de deux listes circulaires. Puis nous faisons correspondre à chacun des éléments de ces listes un des composants graphiques points, de telle sorte que la dernière valeur connue des données corresponde aux deux points les plus à droite de la représentation. Enfin, nous changeons la position relative de ces points pour qu'ils correspondent à la valeur des données qu'ils représentent en leur associant une méthode d'animation correspondant à un déplacement suivant l'axe vertical.

Désirant que l'affichage de ces éléments graphiques se fasse comme pour la représentation et l'animation d'un tableau, en employant si possible tout l'espace de visualisation du module, nous avons ajouté une contrainte à cette animation. Celle-ci est alors chargée d'afficher l'objet graphique proportionnellement à la plus grande des valeurs des deux listes du module.

Pour permettre d'estimer la valeur relative des éléments ainsi visualisés, nous avons ajouté à la représentation deux composants graphiques textuels chargés d'afficher l'échelle des valeurs des données et d'associer à chacun une animation de changement de texte.

Les modules permettant d'implémenter l'algorithme sont présentés dans le tableau 5.2 décrivant plus explicitement les paramètres pouvant intervenir dans l'animation de leur représentation, leur aspect visuel statique, les paramètres et les éléments graphiques associés animés et le type d'animation qui leur a été joint.

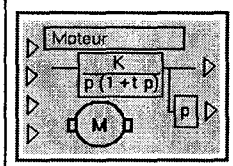
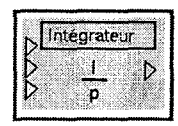
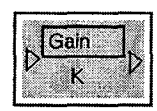
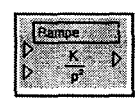
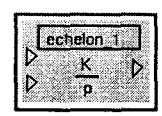
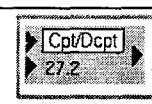
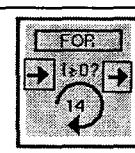
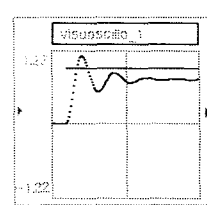
Module	**Paramètres du module**	**Présentation du module**	**Variable animée**	**Elément graphique animé**	**Type d'animation**
Moteur	K, τ, baseDeTemps, signalEnt, signalSortPos, signalSortVit		K, τ	textes	changements de textes
Intégrateur	baseDeTemps, i, signalEnt, signalSort		i	textes	changements de textes
Sommateur	signalEnt+, signalEnt-, signalSort		-	-	-
Gain	k, signalEnt, signalSort		k	texte	changement de texte
Entrée rampe	k, baseDeTemps, signalSort		k	texte	changement de texte
Entrée échelon	k, baseDeTemps, signalSort		k	texte	changement de texte
Variable de sortie	varOut		varOut	texte	changement de texte
Compteur décompteur	varIn, pas varOut		varOut	texte	changement de texte
Boucle "for"	"i"		i	texte	changement de texte
Visualisation de l'historique de deux données	ent1, ent2, sort		ent1, ent2	les points graphiques	déplacements suivant l'axe vertical

Tableau 5.2: *Liste des modules utilisés pour implémenter l'algorithme de traitement d'un asservissement moteur*

Tous ces modules permettent alors d'implémenter un synoptique du système et de visualiser la forme de la réponse pour l'excitation donnée (cf. figure 5.26).

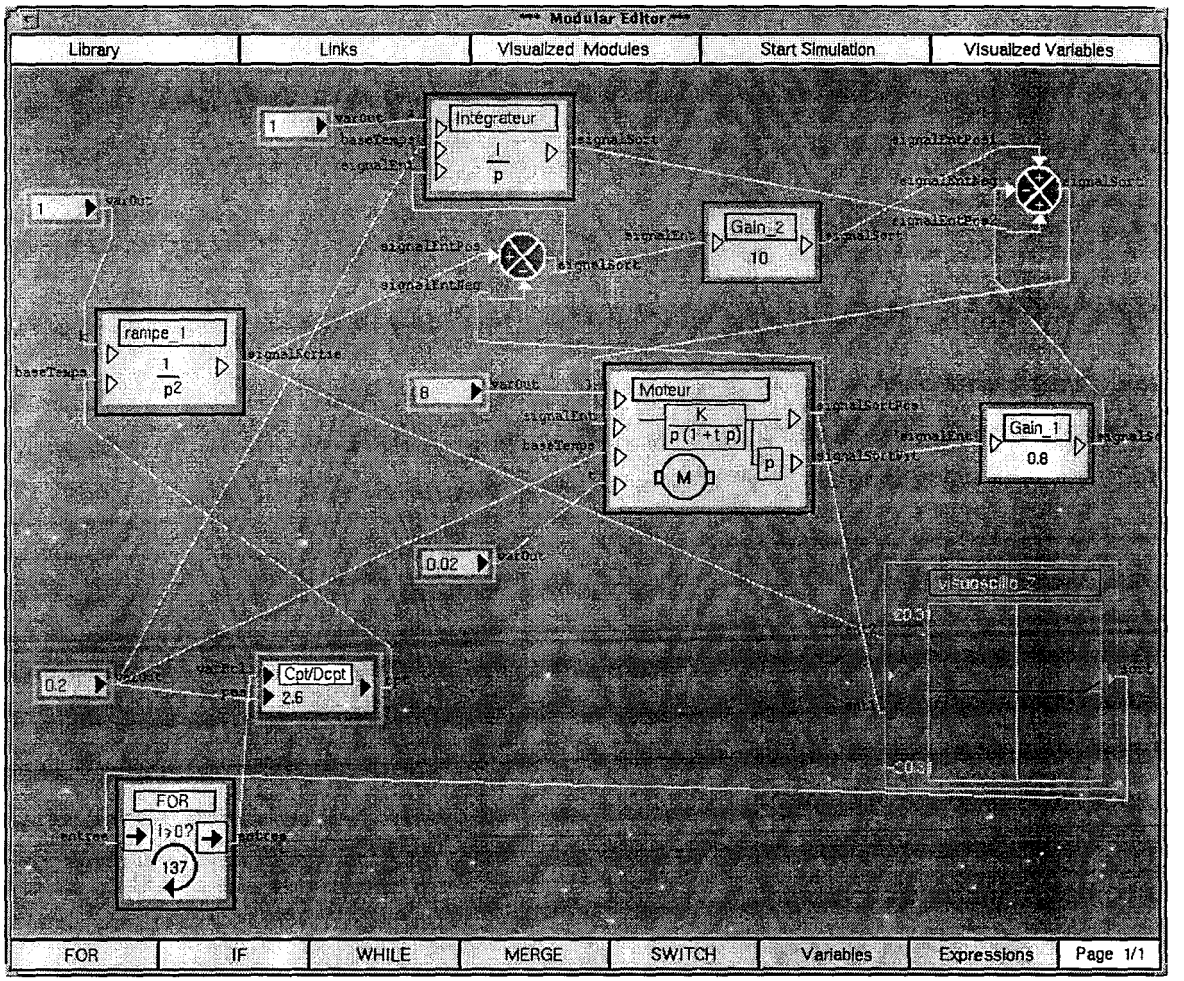
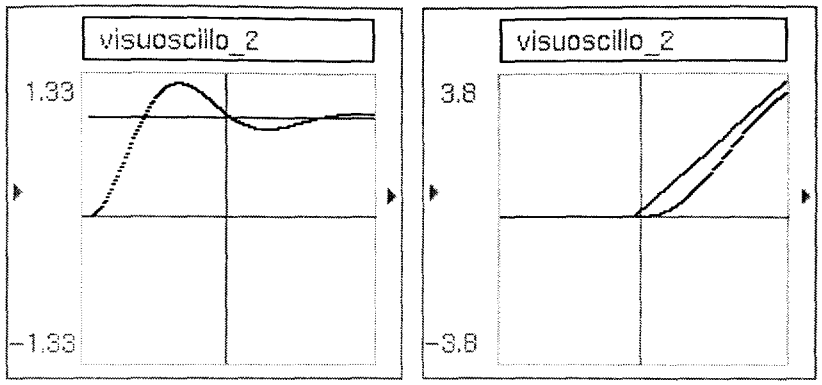


***Figure 5.26** : Implantation d'un programme visualisant la réponse de l'asservissement du moteur à une entrée donnée*

Nous pouvons visualiser à la figure 5.27 la réponse du moteur soumis à une excitation échelon caractérisant l'erreur en position et le dépassement puis une rampe pour l'erreur de traînage.



(a) (b)

Figure 5.27 : Réponses du moteur asservi pour : (a) un échelon, (b) une rampe

Ces courbes sont obtenues avec pour paramètres : un gain de 8 et une constante de temps de 0.8 s pour le moteur, un gain de 1 pour la correction intégrale, un gain de 0.8 pour la correction dérivée et enfin un gain global de 10.

V.5.4 - Conclusion

Cet exemple avec le précédent montre que la plate-forme de programmation visuelle peut simuler des applications provenant de domaines très différents. Ceci est dut à la simplicité du moteur de simulation et à sa souplesse de traitement. Cependant celui-ci présente le défaut de compliquer dans certains cas la tâche de l'utilisateur. Le moteur n'étant pas spécialisé pour une application particulière le contraint parfois à employer des modules connectés de manière particulière pour obtenir le traitement désiré (gestion du temps dans l'exemple ci-dessus).

L'utilisation de modules variables et les différentes interactivités associées à la plate-forme (cf. chapitre IV.7) permettent de tester très rapidement l'influence des différents paramètres de l'asservissement.

Cependant la multiplication des modules au sein de l'application accroît la bibliothèque de telle sorte qu'il devient difficile de retrouver celui désiré sans hiérarchiser le menu permettant de les sélectionner.

Finalement, comparativement à un autre système de programmation visuelle adoptant le principe de flux de données, la plate-forme développée a pour principale particularité de pouvoir associer à chaque module une image animée qui le caractérise et visualiser individuellement chaque traitement interne de manière graphique.

V.6 - CONCLUSION

Ce chapitre montre que notre environnement peut animer d'une part de très larges et très divers algorithmes et d'autre part que par l'intermédiaire de la plate-forme de programmation visuelle celui-ci ne se limite pas à visualiser l'activité d'un algorithme mais permet d'établir de véritables applications.

Grâce aux outils développés et décrits dans les précédents chapitres, nous avons pu écrire les algorithmes indifféremment en Pascal ou en Smalltalk et établir avec une relative facilité une représentation et une animation de leurs informations. Mais ceci n'a été rendu possible que par les différents concepts de convivialité et d'interactivité adoptés, ainsi que la méthodologie de spécification basée sur les relations de dépendance qui sépare totalement ces deux phases.

Cependant, faisant appel en grande partie à l'expérience de l'utilisateur et à son pouvoir d'abstraction, l'établissement d'une représentation de synthèse générique peut être difficile à établir. Aussi, la visualisation des données dans des fenêtres autonomes sans que l'utilisateur n'ait à établir une quelconque construction ou relation peut être particulièrement appréciable. Même si celle-ci n'est pas comparable à une visualisation de synthèse, les données comportent, comme nous l'avons vu dans les deux premiers exemples, intrinsèquement suffisamment d'informations pour obtenir une représentation plus explicite et plus riche que celle que l'on pourrait obtenir sous leur forme textuelle initiale. Si ces représentations ne sont pas en soit suffisantes pour comprendre directement le fonctionnement de tous les algorithmes, la construction d'une visualisation spécifique est toujours possible et demande moins de connaissances et de temps d'élaboration. Ainsi la construction d'une représentation générique fiable demande généralement une journée ou d'avantage et une construction spécifique, au plus, une demi-journée.

Les algorithmes présentés sont certes relativement simples, mais l'enrichissement des bibliothèques de l'environnement peut rapidement contribuer à ce que le système puisse implémenter de nouveaux et très divers algorithmes.

CHAPITRE VI

CONCLUSIONS ET PERSPECTIVES

Un des premiers buts de notre recherche a été de construire un système avec lequel tout programmeur peut obtenir une représentation graphique et une animation des informations d'un algorithme, dans un environnement convivial et interactif. Pour prendre en compte ces deux aspects, nous sommes partis d'un modèle structurel qui nous a aidé à déterminer les composants d'un tel logiciel.

L'étude de ces éléments nous a amené à développer un environnement basé sur le fenêtrage multiple et sur l'élaboration d'un débogueur évolué permettant de visualiser les informations de programmes élémentaires.

Nous pouvons par l'intermédiaire des outils conçus obtenir, dans des fenêtres autonomes, une représentation directe et structurale des variables de l'algorithme ou de toutes expressions appartenant à celui-ci, ainsi que visualiser son comportement par l'entremise d'une visualisation de synthèse qui représente l'abstraction du problème traité informatiquement.

Concrètement, nous nous sommes surtout attachés à développer les outils et les concepts nécessaires pour obtenir une représentation graphique des informations de la majorité des algorithmes dans un environnement interactif et indépendamment de personnes spécialisées dans l'animation ou dans la programmation.

En fait, si nous voulons que ce type de système soit utilisé comme un outil d'aide à la programmation, il ne faut pas que l'effort requis pour obtenir une visualisation dépasse les bénéfices perçus par un utilisateur. Par conséquent, n'importe quelle information susceptible de l'intéresser doit pouvoir être obtenue ou facilement élaborée indépendamment de l'algorithme considéré.

Nous avons ainsi développé une méthode de spécification autonome qui nous permet de régir les événements de la simulation et par là même la remise à jour des visualisations en lançant les animations de manière transparente pour l'utilisateur. La mise en place de ce concept nous permet d'animer tout algorithme dont les informations dites intéressantes sont basées sur une ou plusieurs données et des expressions simples ou complexes ce qui inclue la plupart des algorithmes.

D'autre part, l'utilisation de cette méthode permet d'obtenir une simulation interactive car la conception d'un programme et la représentation des informations sous forme d'animations graphiques peuvent être considérées comme deux processus totalement séparés. Un utilisateur peut alors changer, comme dans la plupart des systèmes, tous les paramètres de visualisation mais aussi modifier au cours de son exécution le code, les données d'un programme et conserver une cohérence avec les objets graphiques animés qui les représentent.

Si notre système fournit pour la plupart des données classiques une représentation propre, il n'est pas concevable d'élaborer une bibliothèque d'animations suffisamment exhaustive pour représenter le comportement de tous les algorithmes. Aussi, pour que l'utilisateur puisse créer ses propres images, nous avons introduit un modèle d'architecture graphique orienté objet baptisé P.Os.T. (Présentation, Objet de simulation, Traducteur). Celui-ci permet alors de créer automatiquement par l'intermédiaire d'une interface interactive et conviviale une représentation et une animation sans connaître de langage graphique spécifique.

Cependant, ne pouvant représenter que des programmes de taille réduite, nous avons introduit une plate-forme de programmation visuelle. Celle-ci permet alors de concevoir graphiquement à partir de modules ou de composants logiciels de larges algorithmes et de visualiser indépendamment les unes des autres les informations qu'ils renferment, par l'intermédiaire des outils développés pour la visualisation de programme.

Pour ce faire, nous avons défini des structures graphiques et conceptuelles telle que chaque module soit un algorithme élémentaire et possède une représentation propre. Nous avons également établi les protocoles pour d'une part, contrôler graphiquement ces

composants et d'autre part constituer les relations inter-modules pour obtenir un programme écrit visuellement sous la forme d'une structure simple et cohérente.

Une telle plate-forme permet non seulement la visualisation de larges programmes par l'introduction des concepts d'abstraction procédurale et de programmation par page, mais également une réutilisation des composants logiciels pour créer de véritables applications. En fait, la hiérarchisation visuelle, l'itération, le flux de contrôle et les expressions basées sur différents paramètres accroissent les paradigmes des flux de données traditionnels rendant ainsi notre système utilisable effectivement dans de nombreux domaines, incluant le contrôle de processus automatique et la simulation de systèmes.

En résumé, nous obtenons un environnement homogène avec les principales fonctionnalités suivantes :

- présentation du traitement d'un algorithme ou d'un module sous sa forme textuelle (écrit en Smalltalk ou en Pascal) avec affichage des expressions en cours d'évaluation ;

- possibilité d'écrire graphiquement et de visualiser de larges[1] programmes par l'intermédiaire de l'outil de programmation visuelle ;

- présentation des structures de données ou des expressions jugées nécessaire à la compréhension et de tout programme dans des fenêtres autonomes sous une forme graphique et animée ;

- présentation du comportement global ou de synthèse d'un programme ou d'un module par l'intermédiaire d'une représentation abstraite. Celle-ci se caractérise par l'adjonction de plusieurs structures de données pour une seule représentation et constitue généralement l'image du module dans l'outil de programmation visuel ;

- altération des structures de données de tout programme c'est-à-dire la valeur d'une variable d'un algorithme ou d'un module ;

- changement du code source d'une méthode, d'une procédure appartenant à un simple algorithme ou à un module.

---

1) Effectuant plus d'une tâche

- et enfin élaboration de manière conviviale, interactive et indépendante des représentations et des animations par l'intermédiaire du modèle P.Os.T. et établissement d'une riche bibliothèque d'animation.

L'approche orientée-objet nous a permise de structurer le système, de le rendre homogène, interactif, et d'utiliser en particulier un de ces points forts, sous la forme de la réutilisabilité de composants (ou modules) pour développer l'outil de programmation visuel.

Nous pouvons considérer que le système obtenu contribue à ce que la représentation graphique et l'animation soient rapidement introduites dans les sciences employant l'algorithmique car il permet de visualiser graphiquement les informations d'un programme simple ou large et ce indépendamment de celui-ci. Concrètement, de plus en plus de scientifiques utilisant l'informatique comme moyen de calcul réalisent que la visualisation peut simplifier les premières approches en apportant un support visuel à leur traitement. Mais pour qu'un tel système soit plus systématiquement implanté dans un environnement de programmation, il est nécessaire que la première préoccupation de toute future étude soit son extension vers une plus grande audience.

Nous envisageons ainsi après avoir développé un traducteur Pascal sous Smalltalk, permettant d'écrire un algorithme indifféremment dans ces deux langages, d'implémenter un traducteur C++ basé sur les mêmes structures.

Actuellement le code d'un algorithme est implémenté et présenté textuellement rendant notre système moins dépendant d'un éditeur spécialisé qui emploie généralement un langage visuel spécifique. Cependant, il serait envisageable de montrer graphiquement leur structure sous forme d'organigrammes ou en employant un autre formalisme visuel, ou peut être de combiner plusieurs paradigmes visuels différents où chacun prendrait le relais là où l'expressivité de l'autre commence à être inefficace.

Une aide à l'écriture des programmes par l'intermédiaire d'un éditeur dirigé par la syntaxe [HAB 86, TEI 81] est envisageable en étant toujours portable dans n'importe quel environnement. Au demeurant, il est souvent reproché aux systèmes de programmation visuelle d'établir des programmes qui ne peuvent être transférés et utilisés sans disposer des outils développés spécifiquement pour les écrire. Il serait donc nécessaire de pouvoir établir une version textuelle de tels algorithmes pour qu'ils puissent être employés et lus par tout informaticien.

Un autre domaine où la recherche n'est encore qu'à ses balbutiements est l'emploi de la visualisation de programmes pour présenter le comportement d'algorithmes parallèles. La plupart des systèmes développés dans ce domaine sont généralement basés sur les mêmes principes de spécification, d'interaction et de convivialité que les logiciels de programmation visuelle traditionnelle et hérite du même coup des mêmes problèmes [KRA 96].

Une évaluation en profondeur du système par un ou plusieurs spécialistes en ergonomie et un échantillon représentatif d'utilisateurs reste également a être effectuée. Celle-ci, en dehors d'identifier les points forts et points faibles de l'application, permettra de déterminer le comportement des utilisateurs face à cet outil et doit prouver qu'il facilite l'évaluation et le développement de nouveaux programmes.

Il serait par ailleurs judicieux de vérifier que l'utilisation du son, de la couleur, de l'animation et de la représentation des objets graphiques en deux dimensions est suffisante pour représenter les comportements de toutes les informations visualisables. Nous pouvons envisager, pour les algorithmes utilisant des données multidimensionnelles, de développer un environnement graphique en trois dimensions (ou plus), avec un outil de navigation dans l'hyperespace.

Du reste, par l'intermédiaire de notre système, il serait intéressant d'explorer l'emploi de différentes formes de représentations de l'information pour définir l'impact psychologique de chacune d'elles dans le processus éducatif d'analyse d'algorithme. Par exemple, évaluer si l'animation graphique correspondant à un changement de forme est plus facilement perçue et comprise qu'un changement de couleur. Ou encore que la visualisation de la variable booléenne sous la forme d'un feu bicolore employant la couleur rouge et verte ne gène en rien dans la compréhension d'un algorithme car ces couleurs correspondent à un code socialement bien déterminé.

Notre système doit maintenant passer en exploitation pour en tester les diverses facettes et évaluer son efficacité. Des améliorations ou des ajouts peuvent sans difficulté y être effectués dus à l'approche orientée-objet employée. Mais le point le plus essentiel est certainement d'enrichir les différentes bibliothèques du système, en tenant compte en particulier de certains secteurs d'application comme le traitement d'image ou le traitement du signal.

Pour conclure, nous pouvons dire que la programmation visuelle et la visualisation de programmes sont des domaines complémentaires très intéressants. Ils sont porteurs d'énormément de promesses pour améliorer le processus de programmation pour les programmeurs comme les non-programmeurs.

Nous sommes convaincu que les techniques graphiques seront demain un vecteur essentiel de la communication avec l'ordinateur.

ANNEXE A

UN TRADUCTEUR PASCAL SOUS SMALLTALK

A.1 - INTRODUCTION

Les algorithmes sont fondamentaux ; ils sont à la fois indépendants du langage dans lequel ils sont énoncés et de l'ordinateur qui les exécute. Prenons pour nous aider une analogie avec la vie quotidienne. Une recette de tarte aux fruits peut être écrite en anglais ou en français. Pourtant, quelle que soit la langue, l'algorithme est fondamentalement le même. Un langage de programmation n'est qu'un moyen pratique d'énoncer un algorithme, et un ordinateur est un simple exécutant. Le langage de programmation et l'ordinateur ne sont que les moyens, le but étant l'exécution de l'algorithme et le déroulement du traitement correspondant.

A.1.1 - Programmes et langages de programmation

Ainsi, l'algorithme doit être énoncé d'une manière telle que le processeur puisse le comprendre et en exécuter les instructions. On dit que le processeur doit être capable d'interpréter les algorithmes, ce qui signifie qu'il doit être capable de :

- comprendre ce que chaque étape signifie,
- faire l'opération correspondante.

Ainsi un algorithme ne peut être exécuté que s'il est exprimé sous une forme compréhensible par le processeur concerné adaptée à la compréhension humaine. Il ne serait pas sans intérêt que les ordinateurs puissent comprendre l'une de ces formes d'expression, en particulier l'anglais ou le français. Malheureusement, ceci reste une utopie pour les raisons suivantes :

- Les langages ont un vocabulaire considérable et des règles grammaticales complexes. La procédure d'analyse est si complexe et si mal comprise que les algorithmes correspondants restent limités à des sous ensembles restreints de la langue.
- L'interprétation d'une phrase dépend non seulement d'une analyse grammaticale mais encore du contexte dans lequel elle apparaît. De nombreux mots ont plusieurs significations qui peuvent être interprétées uniquement par le contexte. Par exemple, l'expression "une drôle de situation" est ambiguë tant que la signification du mot drôle (curiosité ou humour) n'est pas révélée par le contexte sémantique créé par la phrase.

Puisque l'anglais et le français sont trop complexes pour être compris par les ordinateurs, les algorithmes doivent êtres écrits plus simplement dans un langage de programmation (du type Fortran, Cobol, Pascal, C, etc.).

A.1.2 - La présentation des algorithmes

Ayant fait le choix de Smalltalk-80 pour développer notre logiciel, il nous a paru évident, dans un premier temps, d'utiliser sa propre syntaxe pour écrire nos algorithmes. Toutefois, nous nous sommes très vite rendu compte que si nous laissons, uniquement, ce type de langage à l'utilisateur, nous limitons notre application à un nombre restreint de programmeur. Ceci est dû, d'une part, au fait que Smalltalk est un langage objet basé sur plusieurs concepts qui diffèrent fortement de la programmation procédurale. D'autre part la communication entre objets ne ce fait que par messages. Possédant une bibliothèque de base de plus de 3000 messages, ce langage implique une initiation longue.

Nous avons donc choisi de développer un nouveau compilateur pour utiliser, en plus de la syntaxe Smalltalk, la syntaxe du langage Pascal et ainsi pouvoir écrire nos algorithmes et structures de données dans ce langage. Ce choix a été motivé avant tout par le fait que Pascal jouit d'une certaine popularité, et reste suffisamment général pour que nos algorithmes puissent être facilement reproduits et mis en oeuvre dans n'importe quel autre langage de programmation évolué. Cependant, il faut bien comprendre qu'un algorithme est indépendant du langage de programmation utilisé. L'algorithme d'Euclide programmé en Pascal, C, Ada ou Lisp, reste toujours l'algorithme d'Euclide.

A.1.3 - Différentes phases d'un compilateur

Tout compilateur peut être découpé en différentes phases pouvant être regroupées ou non suivant les compilateurs considérés, figure A.1 [AHO 86].

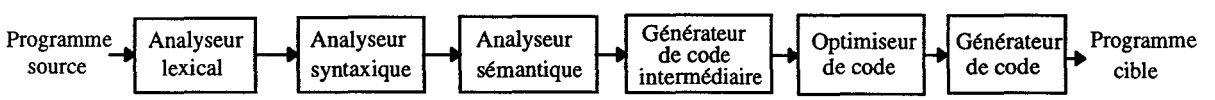


Figure A.1 : Phases d'un compilateur

Dans notre application nous ne développons pas toutes les phases décrites ci-dessus car le "Parser Compiler" de Smalltalk permet, à partir d'une grammaire contextuelle quelconque, de faire une analyse syntaxique et sémantique d'un texte et d'utiliser le générateur de code originel. Ceci est obtenu en redéfinissant simplement et judicieusement le "Parser".

Celui-ci doit pouvoir, à partir de l'analyse de la syntaxe utilisée, vérifier qu'elle est conforme à la définition de la grammaire Pascal (cf. diagramme BNF ou de CONWAY) et de traduire cette définition en un arbre syntaxique Smalltalk approprié, qui devient la machine

virtuelle Pascal. L'arbre ainsi obtenu permet de générer le code intermédiaire (byte-code) de la même façon qu'un texte écrit dans la syntaxe Smalltalk.

Avant de décrire le nouveau "parser" et ces différentes interactions avec son environnement, nous allons présenter la machine virtuelle Pascal chargée de simuler les fonctionnalités de Pascal sous Smalltalk, et de définir ses implications.

Pascal, étant au contraire de Smalltalk un langage typé et procédural, implique de nouvelles considérations. Nous définissons ainsi un nouvel analyseur lexical qui fait appel à une table des symboles comme l'analyseur syntaxique.

A.2 - MACHINE VIRTUELLE PASCAL

Lors de la conception du compilateur Pascal, nous nous sommes heurtés à plusieurs problèmes car la programmation par objets diffère notablement de la programmation procédurale. Cette différence n'est pas tellement sur le plan syntaxique, mais plutôt dans la tournure d'esprit à adopter qui tranche par rapport à celle que l'on aurait pu développer vis-à-vis d'un langage procédural. Dans un langage du type Pascal, une application est formée d'un ensemble de procédures qui effectue des traitements sur les données qu'on leur fournit. Dans la programmation par objets, ce sont les données qui effectuent des opérations sur elles-mêmes, suite à une demande explicitement formulée. Il n'y a donc pas de formes équivalentes entre un langage procédural et un langage objet car les deux concepts sont totalement différents.

A.2.1 - Différences et similitudes entre les langages Pascal et Smalltalk

Une des premières tâches va être de définir ce qu'est pour nous un programme procédural en terme de concept objet. Pour ce faire, nous allons définir succinctement la structure du langage source et du langage cible et comparer ces deux concepts pour nous permettre de déterminer un modèle de structure pour développer la machine virtuelle Pascal.

A.2.1.1 - Structures principales de Pascal

Pascal est un langage procédural à structure de bloc. Une procédure est une partie (ou bloc) de programme qui se suffit à elle-même. Ainsi, la structure d'une procédure ou d'une fonction est identique à celle d'un programme. Elle est constituée :

- d'une entête, permettant de l'identifier (avec ou sans paramètre),
- d'une partie déclarative, qui peut être vide,
- du corps du programme, constitué d'instruction.

La subdivision en blocs imbriqués est par ailleurs une des principales caractéristiques d'un programme Pascal.

Comme chaque bloc peut avoir des déclarations d'objets, il est important de connaître le domaine de validité d'un objet, c'est-à-dire les limites de la zone entre lesquelles il est connu. Pour ce faire, nous allons analyser la structure du programme à travers deux types de représentation, figure A.2.

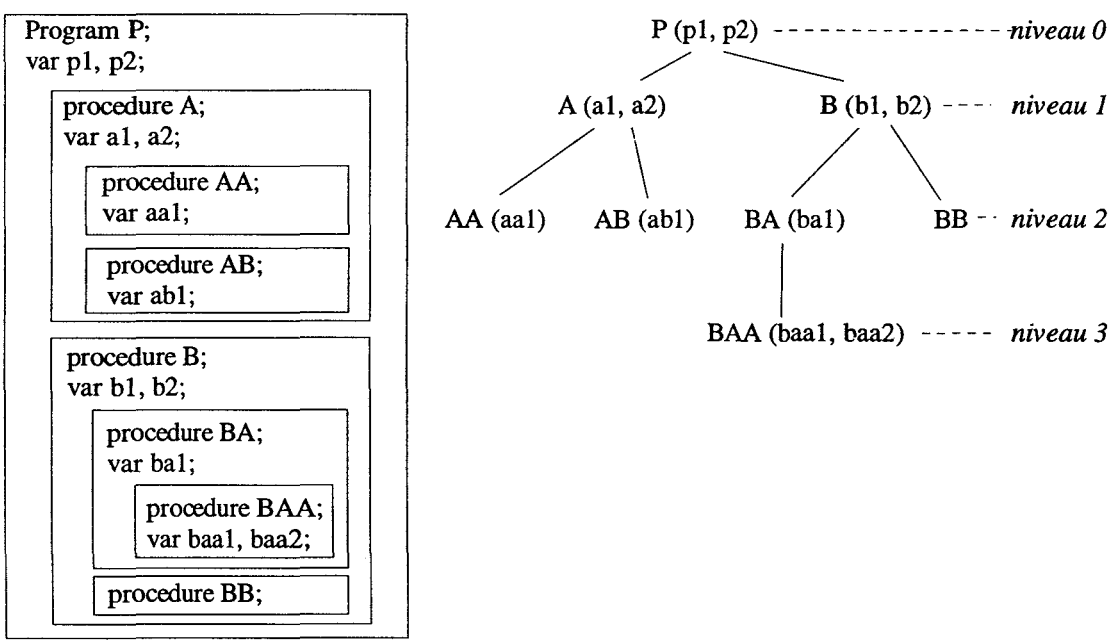


***Figure A.2** : Représentation en blocs imbriqués et en arbre à niveaux d'un programme Pascal.*

Si nous considérons la structure du programme ci-dessus nous pouvons déterminer la portabilité des procédures et des données représenté sous la forme du tableau A.1.

Programme/Sous programme	Procédures utilisables	Objets utilisables [1]
P	A, B	p1, p2
A	AA, AB, A	p1, p2, a1, a2
B	BA, BB, A, B	p1, p2, b1, b2
AA	AA, A	p1, p2, a1, a2, aa1
AB	AA, AB, A	p1, p2, a1, a2, ab1
BA	BAA, A, BA, B	p1, p2, b1, b2, ba1
BB	BA, A, BB, B	p1, p2, b1, b2
BAA	A, BAA, BA, B	p1, p2, b1, b2, ba1, baa1, baa2

***Tableau A.1** : Portabilité des procédures et des données du programme de la figure A.2*

A.2.1.2 - Structure de Smalltalk et comparaison à Pascal

Smalltalk quant à lui est basé sur les concepts objets. Un objet possède une structure de données privée et des procédures privées d'accès et de manipulation de cette structure. Ainsi l'appel d'une procédure privée d'objet ou méthode, passe par l'envoi d'un message à cette procédure.

Une méthode est constituée :

- d'une entête, qui est le nom du message permettant de l'identifier (avec ou sans argument),
- d'une partie déclarative, qui peut être vide (constituée de variables temporaires),
- du corps de la méthode, constitué d'expressions.

Nous voyons qu'une des premières analogies apparentes est de considérer une méthode comme étant l'équivalente d'une procédure ou d'une fonction en Pascal.

[1] Les objets p1, p2, a1, a2 ... sont les mêmes en terme de pointeur.

Mais la question qui reste en suspend est : que peut-être un programme en terme d'objets ?

Car nous ne nous limitons pas à traduire en Smalltalk une simple procédure mais un algorithme complet, celui-ci pouvant être considéré comme un programme à part entière et éventuellement être constitué de plusieurs procédures ou fonctions.

Avant d'aborder ce problème, il nous faut introduire d'autres notions qui ont trait au langage objet. Ainsi un objet, en Smalltalk, peut être constitué de plusieurs sortes de variables dont les principales sont :

- les variables temporaires qui sont utilisées dans le corps des méthodes et sont allouées pour la durée d'exécution de la méthode,

- les variables d'instance qui sont les variables privées d'un objet et sont définies dans sa classe et ses superclasses,

- les variables de classe sont les variables partagées par toutes les instances *d'une seule classe*. Elles sont principalement utilisées pour initialiser des constantes.

Pour présenter le principe de visibilité des méthodes et des variables, nous devons introduire une autre notion importante en objet qui nous permet de mettre en commun des déclarations semblables, appelée héritage. Par exemple, si nous considérons la hiérarchie de la figure A.3. nous pouvons en déduire la portée de chaque méthode et variable (cf. tableau A.2).

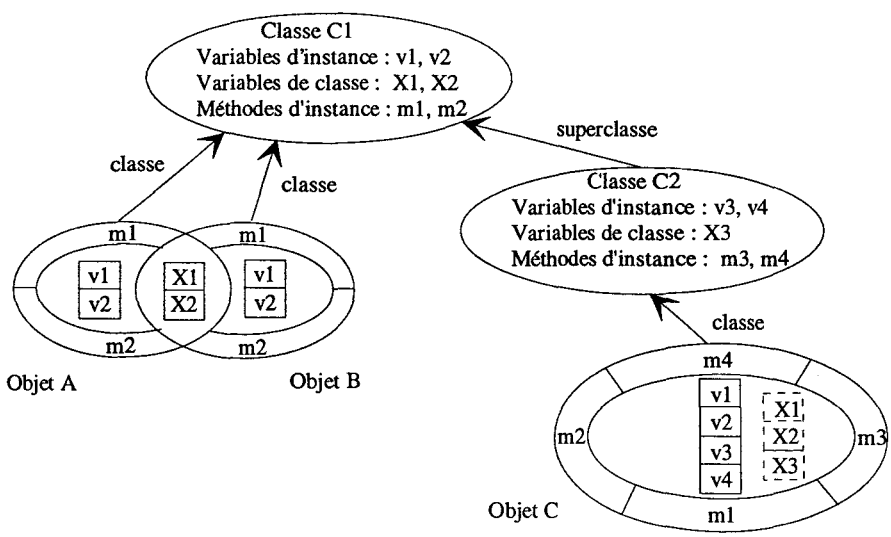


***Figure A.3** : Héritage et instanciation d'objets : C2 est sous classe de C1 ; les objets A et B sont instances de C1 et l'objet C est instance de C2*

Objets	Méthodes utilisables	Variables de classes utilisables [2]	Variables d'instances utilisables [3]
A	m1, m2	X1, X2	v1, v2
B	m1, m2	X1, X2	v1, v2
C	m1, m2, m3, m4	X1, X2, X3	v1, v2, v3, v4

Tableau A.2 : *Portée des méthodes et variables de la figure A.3*

Si nous considérons l'arbre déterminé pour le programme Pascal vu précédemment (cf. figure A.2) et que nous supposons que chaque procédure est un objet, contenant une seule méthode, et chaque variable est une variable d'instance ou de classe, nous obtenons comme tableau de portabilité le tableau A.3.

Objets	Méthodes utilisables	Variables de classes utilisables	Variables d'instances utilisables
P	P	p1, p2	p1, p2
A	A, P	p1, p2, a1, a2	p1, p2, a1, a2
B	B, P	p1, p2, b1, b2	p1, p2, b1, b2
AA	AA, A, P	p1, p2, a1, a2, aa1	p1, p2, a1, a2, aa1
AB	A, AB, P	p1, p2, a1, a2, ab1	p1, p2, a1, a2, ab1
BB	B, BB, P	p1, p2, b1, b2	p1, p2, b1, b2
BA	B, BA, P	p1, p2, b1, b2, ba1	p1, p2, b1, b2, ba1
BAA	BAA, BA, B, P	p1, p2, b1, b2, ba1, baa1, baa2	p1, p2, b1, b2, ba1, baa1, baa2

Tableau A.3 : *Portée des méthodes et des variables pour le programme de la figure A.2 en supposant que chaque procédure est un objet*

Nous nous rendons très bien compte que si les variables de classe ou les variables d'instance (en dehors du problème des pointeurs différents) répondent au critère de visibilité de Pascal, les méthodes n'y répondent pas. Il est de plus impossible de déterminer une hiérarchie pour que les méthodes répondent à ce critère.

---

[2] X1, X2 et X3 sont les mêmes objets.

[3] v1, v2 v3, v4 sont des objets distincts pour chacune des instances inspectées car elles possèdent des pointeurs différents.

A.2.1.3 - Conclusion

Ainsi, une procédure ne peut-être regardée comme un objet simple, inscrit dans une hiérarchie donnée et possédant une seule méthode.

Nous sommes donc amenés à considérer un programme comme un unique objet possédant un ensemble de méthodes. Si nous ne répondons toujours pas au critère de visibilité, basé sur la règle de "l'englobant le plus imbriqué", nous pouvons disposer de toutes les procédures à tout instant.

Ainsi, nous avons choisi de reporter le problème, de vérifier si l'appel d'un sous-programme dans le corps d'une méthode est autorisé ou non, sur le compilateur. Celui-ci doit alors conserver une information sur la portée statique de chaque procédure employée.

Si on considère un programme comme un objet unique, une variable de classe ou d'instance ne peut se comporter comme une variable d'une procédure car elle est forcément partagée par toutes les méthodes de la classe. Nous sommes donc obligés d'opter pour une autre stratégie. Mais avant de présenter le modèle adopté, nous devons considérer un autre problème de compatibilité qui fait appel à une solution analogue.

Ainsi, l'identificateur d'un bloc peut avoir un ou plusieurs paramètres. En Pascal nous avons deux grandes classes de paramètres, ceux passées par valeur et ceux passées par variables ou par références (var). En Smalltalk, un argument ne peut être passé que par valeur (exceptés les objets qui pointent sur d'autres objets, par exemple : l'objet tableau).

D'autre part, nous pouvons observer qu'une variable temporaire en Smalltalk se comporte comme une variable locale en Pascal car elle n'est allouée que pour la durée d'exécution de la méthode, ce qui nous suggère une première solution.

A.2.2 - Présentation du premier modèle

La conception du premier modèle, s'appuyant sur cette dernière remarque, consiste à regarder les variables de chaque bloc comme des variables temporaires. Pour exporter ces variables lors de l'appel d'une procédure nous les mettons en argument du message. Mais comme ces paramètres doivent être passés par variable nous les enregistrons préalablement dans un tableau nous permettant un pointage indirect sur celles-ci (cf. Annexe B).

Nous pouvons également adopter le même principe pour les paramètres de la procédure passés par variable et ainsi agrandir le nombre de paramètres du tableau.

A la fin du traitement d'une procédure, nous devons réaffecter toutes les valeurs de ces variables car elles sont susceptibles d'avoir évolué. Ceci est dû au fait que les données ont été modifiées lors d'une affectation, dans la méthode appelée, en accédant aux différentes valeurs qu'elles représentent par l'intermédiaire du tableau. Ainsi, si nous pointons toujours sur le même objet (tableau) entre l'appel d'une méthode et le retour à l'appelant, nous ne pointons plus sur les mêmes variables (dans la méthode appelée). En effet, l'affectation de ces données s'exprime en terme de changement de pointeur à l'intérieur du tableau mais pas dans la méthode appelante. De même, les données du tableau sont ultérieurement récupérables, pointant toujours sur le même argument.

Etant de même nature, les paramètres passés par valeur sont ajoutés comme de simples arguments du message (cf. Annexe B).

Si le principe énoncé ci-dessus satisfait à tous les critères de portabilité des variables et a été implanté dans notre système, celui ci pose deux grands problèmes :

- Si nous avons des procédures dont l'imbrication est profonde avec énormément de variables, nous aurons un nombre d'arguments énorme à gérer (ayant à chaque appel de procédure à reconstituer le tableau de toutes les variables globales ayant des objets distincts).

- L'ajout de code pour récupérer les paramètres passés par variables, après l'appel de la procédure, peut engendrer une incompréhension de fonctionnement pour l'utilisateur visualisant le traitement de cette méthode dans un "débogueur".
  Ainsi, si on s'attache à visualiser dans un inspecteur une de ces variables, il s'écoule un certain temps où l'utilisateur peut observer une valeur impropre. Ceci est dû au traitement interne des expressions entre le retour du traitement de la procédure invoquée et l'évaluation de la prochaine expression.

Ces inconvénients nous ont amenés à développer une autre approche bien que ce premier modèle soit sémantiquement correct.

A.2.3 - Présentation du second modèle

Le premier modèle considère que chaque variable du programme source Pascal a un équivalent direct en Smalltalk, par exemple une instance de la classe "Integer" en Smalltalk pour un entier, une instance de "Character" pour un caractère, etc. Dans ce cas, l'accès à la donnée est simple et peut utiliser toutes les méthodes (ou opérations) de la classe dont elle est instance.

Mais nous avons vu, précédemment, que pour passer un argument par variable et qu'il soit considéré comme tel, il faut le passer par un objet permettant de ne pas le référencer directement (ce qui est réalisé par l'emploi de l'objet "Array").

Cette constatation nous a amenés à adopter un modèle d'accès indirect aux données. Ceci implique que nous allons construire un objet spécifique pour chaque type de données traité en Pascal, dont une de ces variables d'instance pointe indirectement sur l'objet qu'il est censé représenter, figure A.4.

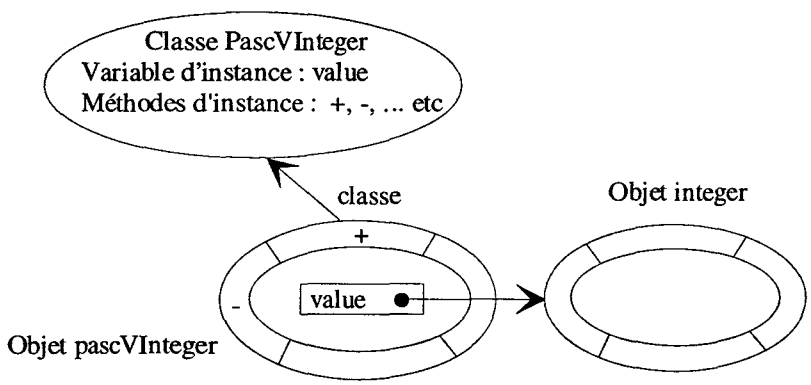


Figure A.4 : Exemple d'accès indirect à la donnée du type "Integer"

Ce modèle implique de redéfinir les méthodes qui sont applicables à cette instance car nous allons envoyer les messages à l'objet "pascVInteger" et non à sa variable d'instance "value" dû au principe d'encapsulation.

Toutes les instances employant ce modèle sont alors regardées, si elles sont passées en argument, comme des objets, indirectement passés par variables. Ceci est dû au fait que, si nous ne pouvons réaffecter l'objet instance de la classe "PascVInteger" passé en argument dans une méthode, nous pouvons assigner une nouvelle valeur à sa variable d'instance, par l'envoi d'un message approprié. En outre, l'accès indirect nous permet de récupérer, après l'appel d'une méthode, la valeur de l'objet réaffecté sans traitement supplémentaire. Car, comme nous l'avons vu précédemment, les valeurs associées aux noms locaux, lorsque le

contrôle retourne d'une activation de procédure, pointent toujours sur le même objet tandis que sa variable d'instance peut pointer sur un objet différent.

Toutefois, toutes les variables adoptant le même modèle fait que nous ne pouvons avoir que des objets passés par variables, s'ils sont en argument d'une méthode.

Ceci implique que pour avoir des paramètres par valeur nous devons dupliquer l'objet référencé et ainsi avoir un pointeur différent lorsqu'il est placé en argument. Dans cette hypothèse, Smalltalk gère bien cette variable comme étant locale à la méthode car pointant sur un objet différent. Mais Smalltalk n'empêche aucunement, en théorie, ayant toujours en paramètre un objet à accès indirect, de réaffecter cette donnée. Ceci est contraire aux règles de Pascal. De ce fait, c'est au compilateur de vérifier que cette variable, passée en argument par valeur, ne soit pas assignée par un objet. Pour ce faire, le compilateur doit conserver une information sur la portée statique de cette donnée.

A.2.3.1 - Création d'une pile

Si le concept énoncé ci-dessus résoud le problème de la portabilité des variables et peut donc être satisfaisant dans son principe, il implique de gérer un flot de paramètres importants lors d'appels de blocs imbriqués. Ceci est dû au fait que, chaque variable considérée comme globale à la procédure appelée, ainsi que ces paramètres, doit être placée en argument du message.

Pour résoudre ce problème nous nous sommes inspirés de Pascal qui gère ces variables dans une pile. Ainsi, lors de l'appel d'une procédure, Pascal empile les nouvelles variables déclarées et dépile celles-ci lorsque le contrôle retourne à l'appelant.

Nous avons donc créé un objet qui est fondé sur la même idée, mais dont les différences se situent sur plusieurs plans.

Ainsi, nous n'empilons pas des variables mais un objet pour chaque bloc traité, dont la structure a entre autres paramètres, deux listes indexées contenant respectivement les variables locales et les paramètres de la procédure. Si nous construisons cette pile comme une variable d'instance du programme, celle-ci étant considérée comme globale, nous pointons quelle que soit la procédure traitée sur le même objet. Dès lors, le dépilement doit être obligatoirement explicité.

Par conséquent, pour nous affranchir de cette obligation, nous allons créer une nouvelle pile pour chaque procédure appelée qui est référencée dans la méthode comme une variable locale. Celle-ci est construite à partir de la pile précédente qui a été passée en argument dans le message de la méthode et est initialisée avec ces nouveaux arguments et ces

nouvelles variables. Etant locale, cette pile n'est prise en compte que pendant le traitement de cette méthode. De ce fait, nous récupérons son ancien état après reprise du contrôle de l'activation de la procédure, ce qui correspond implicitement à un dépilement. D'autre part, ayant ces variables en accès indirect, la pile récupère la nouvelle valeur des objets affectés (cf. Annexe B).

Pour accéder à une de ces données, la méthode doit se référencer au niveau d'enregistrement de la variable dans la pile. Ainsi, nous référençons cette donnée suivant la profondeur de la procédure contenant sa déclaration effective et suivant la place occupée dans la collection des variables ou des paramètres [4]. Si nous prenons l'arbre à niveau de la figure A.2. et que nous voulons référencer, par exemple, la variable b2 lors du traitement de la procédure BA, nous devons avoir un message du type:

pile **niveau:** 1 **varRef:** 2

A.2.4 - Autres problèmes

A.2.4.1 - Le tas

Le problème que nous allons évoquer, maintenant, pour l'obtention d'une machine virtuelle Pascal plus complète est l'affectation dynamique de l'espace mémoire. En Pascal, celle-ci est appliquée au tas pour les données du type pointeur avec l'opérateur "new". La différence entre le tas et la pile se situe sur la conservation des valeurs des noms locaux lorsqu'une activation se termine. Ainsi une activation appelée survit à son appelant.

L'allocation dans le tas partage la mémoire en morceaux contigus en fonction des besoins. Les morceaux peuvent être libérés dans un ordre quelconque ; donc, la plupart du temps, le tas est formé de zones dont certaines sont libres et d'autres occupées. Cet espace mémoire devant être de plus libéré explicitement.

Nous allons donc créer un objet référencé par une variable d'instance qui sera globale pour tout le programme et qui sera chargée de simuler les fonctionnalités d'un tas. Nous n'allons pas nous étendre sur son implémentation mais nous pouvons dire que celle-ci se

[4] Pour garder une homogénéité et simplifier la traduction, le traducteur est amené à enregistrer les données du programme en adoptant le même procédé.

comportera globalement comme un tableau dont certains éléments ne pointent sur aucune donnée lors d'une libération explicite.

A.2.4.2 - Les fonctions

Nous avons surtout parlé du problème des procédures car une fonction n'est qu'une procédure particulière qui retourne une valeur. En Smalltalk toutes les méthodes retournent par défaut le receveur du message. Si l'on veut que la méthode retourne le résultat d'une expression spécifique, on fait précéder cette expression du symbole "^". Dès qu'une expression précédée de ce symbole est évaluée l'objet résultant est renvoyé à l'expéditeur du message ; les expressions qui la suivent sont alors ignorées.

En Pascal l'assignation de la fonction n'empêche nullement celle-ci de traiter toutes les expressions qui la suivent avant de retourner cette valeur. De ce fait, nous avons choisi de construire la fonction de la manière suivante[5] :

en Pascal :

```
function undeplus (i : integer ) : integer;
begin
        statement1;
        undeplus := i + 1;
        statement2
end;
```

en Smalltalk :

```
undeplus: i

| undeplus |

statement1.
undeplus := i + 1.
statement2.
^undeplus
```

En Pascal le nom de la fonction devient une variable en Smalltalk et la valeur est renvoyée à la place du receveur à la fin de la méthode.

A.2.4.3 - Bibliothèque de procédures et de fonctions

Pour simplifier l'interprétation et la génération d'une bibliothèque de procédures et de fonctions du type Turbo-Pascal, nous avons choisi d'enregistrer celles-ci dans un dictionnaire, pointé par une variable d'instance. Ainsi ce dictionnaire peut être initialisé à la demande et disponible pour tout le programme. Il a pour clé le nom de chacune des fonctions Pascal et pour valeur un bloc qui est un objet particulier en Smalltalk. Ainsi un bloc peut contenir une

[5] Pour faciliter la compréhension, la traduction obtenue ci-dessus est effectuée avec des variables à accès direct.

simple expression ou un ensemble d'expressions dont l'évaluation doit être demandée explicitement. L'évaluation d'un bloc renvoie la valeur retournée par la dernière expression du bloc. Il est de plus, possible de passer un ou plusieurs arguments à un bloc avant son exécution. Ceci nous permet de construire nos propres macros chargées de simuler les fonctions et procédures de Pascal.

Exemple de l'interprétation de la fonction *length* en Pascal qui a pour rôle de renvoyer la longueur d'une chaîne de caractères. En Smalltalk nous aurons :

procPool at: #stdProcFun.length put: **[:arg | arg put: arg size]**

#stdProcFun.length, est la clé de la fonction length dans le dictionnaire "procPool" et le message "size" de Smalltalk renvoie la taille de l'argument.

A.3 - LE TRADUCTEUR PASCAL

Après avoir présenté succinctement le modèle adopté pour la machine virtuelle Pascal, nous allons définir les différentes phases du compilateur basées sur les spécificités du compilateur Pascal originel.

A.3.1 - Le compilateur Pascal

Le premier compilateur [WIR 71] ainsi que le second [AMM 81, 77] produisent du code machine absolu pour les calculateurs de la série CDC 6000. Le second compilateur conduisit au compilateur Pascal-P, qui produit du code appelé P-code pour une machine à pile abstraite [NOR 81]. Mais celui-ci impose des restrictions sévères sur la qualité du code produit et souffre de besoins en mémoire relativement importants ; car le code du corps de chaque procédure est compilé en mémoire principale et écrit d'un seul tenant en mémoire secondaire.

Chacun de ces compilateurs fonctionne en une passe et est organisé autour d'un analyseur par descente récursive.

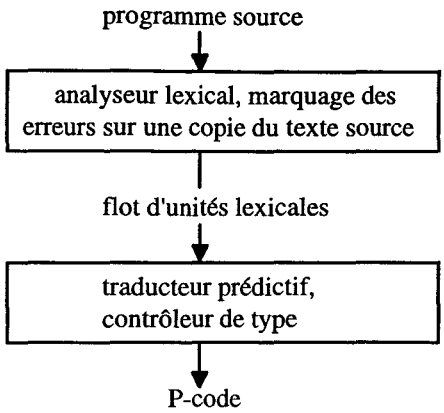


***Figure A.5 :** Phases du compilateur Pascal-P*

La mémoire est organisée en quatre zones :

- le code de la procédure,
- les constantes,
- une pile pour les enregistrements d'activation et
- un tas pour les données allouées en appliquant l'opérateur "new".

Puisque les procédures peuvent être imbriquées en Pascal, l'enregistrement d'activation d'une procédure contient à la fois un lien d'accès et un lien de contrôle. Un appel de procédure est traduit en une instruction de la machine abstraite "marquer la pile", qui prend en argument ces deux liens. Le code d'une procédure adresse l'emplacement mémoire d'une variable locale en utilisant un déplacement par rapport à la fin de l'enregistrement d'activation. L'adressage des variables non locales utilise un couple composé du nombre de liens d'accès à suivre et d'un déplacement. Le premier compilateur employait un adresseur pour l'accès efficace aux variables non locales.

Ainsi, utilisant un langage typé et une machine à pile, nous définissons un nouvel analyseur lexical qui doit interagir avec une table des symboles. Celle-ci est chargée de conserver en mémoire différentes informations comme la portée statique des données. Cette table n'existe pas en Smalltalk étant un langage non typé et à accès direct. Ces phases ont ainsi été développées dépendamment du modèle adopté précédemment.

De plus, la compilation étant effectuée en une passe implique de confondre ces phases en une seule.

A.3.2 - Analyse lexicale

L'analyse linéaire est appelée analyse lexicale. L'analyseur lexical constitue la première phase d'un compilateur. Sa tâche principale est de lire les caractères d'entrée et de produire comme résultat une suite d'unités lexicales que l'analyseur syntaxique va utiliser.

A.3.2.2 - Rôle de l'analyseur lexicale

L'analyseur lexical doit reconnaître les nombres, les opérateurs, les mots clés .. Lorsqu'il reconnaît un objet, il génère une indication sur le type de cet objet (mot, parenthèse, ':' → "comma", etc.), cette indication s'appelle dans notre compilateur un "tokenType". Cette génération est facilitée par la reconnaissance de caractères clés constituant a eux seuls des unités lexicales, tel que les parenthèses, les crochets, le point virgule, le signe plus, etc., et qui sont définis dans une table des symboles appelée "typeTable", figure A.6.

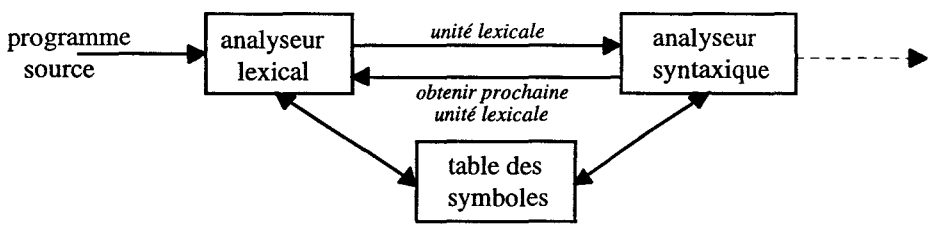


Figure A.6 : *Interaction entre un analyseur lexical et un analyseur syntaxique*

Etant donné que l'analyseur lexical est la partie du compilateur qui lit le texte source, il peut également réaliser certaines tâches secondaires, pour améliorer l'interface avec l'utilisateur. Une de ces tâches est l'élimination, dans le programme source, des commentaires et des espaces qui apparaissent sous la forme de caractères blancs, tabulations ou fin de ligne. Une autre tâche consiste à relier les messages d'erreurs issus du compilateur en indiquant, par un pointeur, l'emplacement dans le programme source de cette erreur.

Il est à noter que pour reconnaître certaines constructions du langage de programmation, nous sommes obligés de lire le caractère en avant, au-delà de la fin d'un lexème avant de pouvoir déterminer avec certitude la bonne unité lexicale (différence entre le ':' et le ':=').

A.3.2.2 - Unités lexicales, modèles et lexèmes

En général, il y a un ensemble de chaînes en entrée pour lesquelles la même unité lexicale est produite en sortie. Cet ensemble de chaînes est décrit par une règle appelée "modèle" associée à l'unité lexicale. On dit que le modèle filtre chaque chaîne de l'ensemble. Un lexème est une suite de caractères du programme source qui coïncide avec le modèle d'une unité lexicale (cf. tableau A.4).

Unité lexicale	Lexèmes	Description informelle des modèles
word	0..3.14..compte..d1	lettre ou chiffre suivi de lettres ou chiffres
operator	> < >= <= <> > >=	> < >= <= <> > >=
leftParenthesis	(	(
rigthBracket	]	]
comma	:	:

Tableau A.4 : *Coïncidence entre le modèle d'une unité lexicale et ses lexèmes*

A.3.2.3 - Erreurs lexicales

Peu d'erreurs sont détectables au seul niveau lexical, car un analyseur lexical a une version très localisée du programme source. La seule erreur possible étant qu'aucun de ces modèles d'unités lexicales ne filtre le préfixe du reste du texte d'entrée.

Dans notre compilateur ce type d'erreur ne peut être produit car nous avons associé à chaque caractère une valeur dans la table "typeTable" qui prédétermine la production du modèle. Ainsi la première unité lexicale du tableau, ci-dessus, est déterminée par tous les caractères qui ont pour valeur "xLetter".

A.3.2.4 - Attribut des unités lexicales

Quand plus d'un modèle reconnaît un lexème, l'analyseur lexical doit fournir, aux phases suivantes du compilateur, des informations additionnelles sur le lexème reconnu. Par exemple "word" correspond à la fois aux chaînes 0 et 1.5, mais il est essentiel, pour le générateur de code, de savoir quelle chaîne a effectivement été reconnue et également de savoir de quel type est ce nombre (entier ou réel).

Nous renvoyons donc le lexème effectivement reconnu dans une variable d'instance nommée "token" et l'analyseur lexical réunit les informations sur les unités lexicales dans des

attributs qui leur sont associés grâce à la table des symboles. Ainsi, à chaque unité lexicale reconnue, nous lui associons son attribut dans une variable d'instance nommée "tokenAttr".

Les attributs influent sur la traduction des unités lexicales, et les unités lexicales influent sur les décisions d'analyse syntaxique.

A.3.3 - Table des symboles

Une structure de données appelée table de symboles est utilisée pour ranger des informations sur les diverses constructions du langage source. Par exemple, au cours de l'analyse syntaxique, la chaîne de caractères ou lexème composant un identificateur peut ajouter à cette entrée des informations comme le type de l'identificateur, son utilisation (par exemple procédure, variable ou étiquette) et son adresse en mémoire ou la place qui lui est réservée dans la pile de la machine virtuelle. Ainsi, cette table est initialisée au cours de l'analyse de la partie déclarative du programme, des procédures et fonctions utilisées.

Celle-ci est en fait, dans notre compilateur, une liste qui fonctionne comme une pile et qui a pour éléments non pas des variables mais des "blockAttr". Ainsi à chaque bloc Pascal (défini dans l'Annexe A.2.1.1) nous ajoutons un "blockAttr" qui contient, entre autres paramètres, un dictionnaire. Celui-ci a pour clés les variables locales de la procédure ou de la fonction ; et à chacune de ces clés nous associons l'attribut approprié à ces variables. Ce principe nous permet de résoudre le problème de la portabilité d'une variable. Car après la fin de la compilation d'une procédure ou d'une fonction, nous enlevons systématiquement le "blockAttr" associé, étant alors au sommet de la pile. De ce fait, les différentes variables associées à cette procédure ou fonction ne sont plus accessibles, car elles n'apparaissent plus comme des références possibles dans la table.

A.3.4 - Analyse syntaxique

L'analyse hiérarchique est appelée analyse syntaxique (ou quelque fois analyse grammaticale). Elle consiste à regrouper les unités lexicales du programme source en structures grammaticales qui sont employées par le compilateur pour synthétiser son résultat. La structure hiérarchique d'un programme est alors exprimée par des règles récursives.

Tout langage de programmation a des règles qui prescrivent la structure syntaxique des programmes bien formés. En Pascal, un programme est formé de blocs, un bloc d'instructions, une instruction d'expressions, une expression d'unités lexicales et ainsi de suite.

Ainsi la syntaxe du langage Pascal est décrite par une grammaire dite non contextuelle ou notation BNF (Backus-Naur Form), c'est à dire que la grammaire est formée de terminaux, de non-terminaux, d'un axiome et de productions.

- Les terminaux sont les symboles de base à partir desquels les chaînes sont formées. L'expression unité lexicale est synonyme de terminal quand on parle de grammaire pour les langages de programmation. Ainsi, chacun des mots clés, "if", "then" et "else" est un terminal.

- Les non-terminaux sont des variables syntaxiques qui dénotent un ensemble de chaînes. Ils imposent une structure hiérarchique sur le langage qui est à la fois utile pour l'analyse syntaxique et pour la traduction.

- L'axiome est un non-terminal particulier tel que l'ensemble des chaînes qu'il dénote est le langage défini par la grammaire.

- Les productions de la grammaire spécifient la manière dont les terminaux et les non terminaux peuvent être combinés pour former des chaînes.

Exemple : *expr* := *expr op* + *expr*
expr := (- *expr*)
expr := id

Dans cette grammaire les symboles terminaux sont : ( ) + - id , les non-terminaux sont *expr* et *op* et l'axiome est *expr*.

A.3.4.1 - Rôle de l'analyseur syntaxique

Dans notre modèle de compilateur, l'analyseur syntaxique obtient une chaîne d'unités lexicales de l'analyseur lexical, comme illustré à la figure A.7, et vérifie que la chaîne peut être engendrée par la grammaire du langage source. Nous supposons que l'analyseur syntaxique signale chaque erreur de syntaxe de façon intelligible.

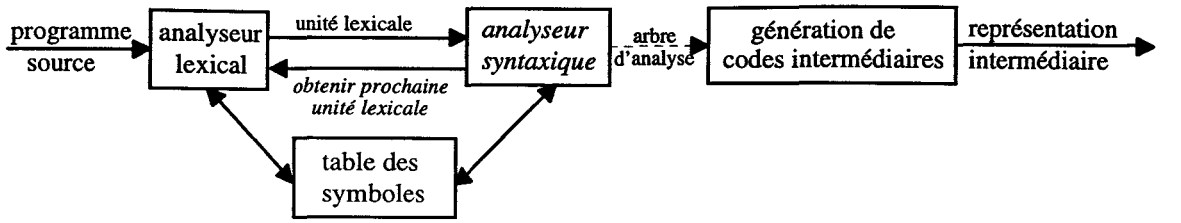


Figure A.7 : Emplacement de l'analyseur syntaxique dans le compilateur

L'interaction entre l'analyseur syntaxique et l'analyseur lexical est réalisée de façon simple en faisant de l'analyseur lexical une méthode appelée par l'analyseur syntaxique qui rend les unités lexicales une par une, à la demande.

D'autre part, un certain nombre de tâches sont effectuées au cours de l'analyse syntaxique, comme la mise en table des symboles d'informations sur les différentes unités lexicales, le contrôle des types et autres sortes d'analyse sémantique.

A.3.4.2 - Analyse syntaxique prédictive

La grammaire non contextuelle de Pascal n'ayant aucune récursivité à gauche peut employer l'analyse syntaxique par descente récursive qui est une méthode d'analyse syntaxique descendante dans laquelle on exécute un ensemble de méthodes récursives pour traiter l'entrée, une méthode étant associée à chaque non-terminal d'une grammaire.

Cette analyse est préférée à l'analyse descendante simple, car la sélection d'une production pour un non-terminal est un processus d'essai-et-erreur, c'est-à-dire qu'on peut avoir à essayer une production et à rebrousser chemin pour essayer une autre production si la première s'avère ne pas convenir. Et l'analyse ascendante est très complexe à mettre en oeuvre, cependant, elle peut manipuler une classe plus large de grammaire.

Ici, on présente une forme spéciale d'analyse syntaxique par descente récursive appelée analyse syntaxique prédictive, où le symbole de prévision détermine d'une manière non ambiguë la méthode à choisir pour chaque non-terminal. L'unité lexicale qui vient d'être reconnue dans le flot d'entrée est souvent appelée symbole de prévision. La suite des méthodes appelées au cours du traitement de l'entrée définit implicitement l'arbre syntaxique de l'entrée.

Exemples : *instr* := **if** *expr* **then** *instr* **else** *instr*
(ou) **while** *expr* **do** *instr*
(ou) **begin** *liste_instr* **end**

Les mots clés "if", "while" et "begin" nous (pré)disent quelle alternative est seule susceptible de réussir lorsque l'on développe une instruction.

Ainsi, pour chaque construction Pascal, l'analyseur syntaxique produit une construction Smalltalk équivalente ou génère une construction spécifique pour obtenir un fonctionnement similaire. Par exemple, pour permettre de simuler l'interaction des données avec son environnement qui est comme nous l'avons vu précédemment dépendant de la machine virtuelle Pascal choisie.

Certaines constructions en Pascal ont une équivalence simple en Smalltalk.

Par exemple, en Pascal :	*La traduction en Smalltalk est :*
if *boolean-expression* **then** *statement1* **else** *statement2*;	*boolean-expression* **ifTrue:** [*statement1*] **ifFalse:** [*statement2*].

D'autres constructions peuvent appliquer une transformation syntaxique plus importante et peuvent, également, avoir plusieurs transformations sémantiquement correctes.

Par exemple, en Pascal :	*La traduction en Smalltalk est :*
for i := 1 **to** max **do** *statement1*;	i := 1. [i <= max] **whileTrue:** [*statement1*. i := i + 1]. ou [6] : 1 **to:** max **do:** [:i | *statement1*].

Certaines fonctions telles que le "or" le "and", etc., peuvent être traduites, également, de deux manières différentes :

En Pascal :

expr1 **or** *expr2* ⇒ *expr1* et *expr2* sont toujours évalués

[6] La première traduction est préférée pour des raisons d'optimisation de code en Smalltalk. Car étant un langage semi-interprété, Smalltalk est basé sur un nombre de "byte-code" prédéfinis, dont l'utilisation direct accélère le traitement d'une méthode (ce que fait le message "whileTrue:" mais pas le massage "to: do:").

En Smalltalk :

expr1 **|** (*expr2*)	⇒	*expr1* et *expr2* sont toujours évalués
expr1 **or:** [*expr2*]	⇒	*expr1* est évalué ; s'il renvoi l'objet "true", *expr2* n'est pas évalué.

Ainsi, pour que l'évaluation d'une condition en Pascal soit sémantiquement correcte, nous choisissons la première traduction.

D'autres problèmes sont liés au type de données manipulées. Par exemple, en Pascal les tableaux peuvent être indicés par des entiers dans un interval fermé quelconque, par exemple, entre -x et +y. En Smalltalk, nous n'avons que des indices du type positif dont le premier indice est obligatoirement 1.

Nous sommes, de ce fait, amenés à gérer un index de référence lors de la traduction.

Il est à noter que les limites respectives de la taille du tableau dépendent de la place mémoire disponible pour les deux langages considérés.

Nous n'allons pas présenter, pour des raisons de places et d'intérêts, toutes les transformations possibles de chaque construction grammaticale du langage Pascal.

Mais, au cours de cette recherche d'équivalence, nous avons répertorié plusieurs niveaux de conversion qui résument, parfaitement, les différents types de transformation possibles que doit effectuer notre analyseur syntaxique.

- Les changements simples sont les substitutions lexicales ; les opérateurs et certains mots clés en Pascal sont remplacés par les opérateurs et "token" correspondant en Smalltalk.
  Par exemple le mot clés "if" est remplacé par la chaîne de caractère nul, l'opérateur "and" par "&", etc.

- Le second niveau de traduction est appelé niveau de transformation syntaxique. Certaines constructions en Pascal ont une construction équivalente en Smalltalk, mais avec des ordres différents dans les "tokens" (avec la possibilité d'une substitution). Un simple exemple est la construction d'une expression booléenne avec la fonction "not" en Pascal qui est de la forme :

  not *expression1*;

L'équivalent en Smalltalk est l'ordre inverse.

expression1 **not**.

- Un troisième niveau est la transformation dépendante du contexte. Car nous avons vu précédemment que pour obtenir l'équivalent des paramètres du code Pascal par valeur et par variable, étant donné qu'en Smalltalk nous n'avons que des paramètres par valeur, nous générons un code approprié pour reproduire une gestion des données analogues.

- Un quatrième niveau de traduction est l'expansion sémantique due par exemple au "with" qui n'a pas d'équivalent en Smalltalk, de l'accès à une donnée par l'intermédiaire de la pile appelée "stack" ou de l'initialisation de celle-ci, etc.

- Enfin la transformation finale est l'appel des fonctions particulières de Pascal par la création d'une librairie (introduite dans l'Annexe A.2.4.3). Ces fonctions peuvent faire appel à une similitude simple en Smalltalk faisant appel à une expression ou elles peuvent faire appel à une macro.

A.3.4.3 - Traitement des erreurs syntaxiques

La plupart des spécifications des langages de programmation ne décrivent pas la réponse des compilateurs aux erreurs ; celles-ci sont laissées au concepteur du compilateur.

Nous savons que les programmes peuvent contenir des erreurs à différents niveaux. Par exemple, les erreurs peuvent être :

- lexicales, comme l'écriture erronée d'un identificateur, mot clé ou opérateur ;

- syntaxiques, comme une expression arithmétique mal parenthèsée ;

- sémantiques, comme un opérateur appliqué à un opérande incompatible ;

- logiques, comme un appel récursif infini.

Beaucoup d'erreurs sont par nature syntaxiques ou sont révélées lorsque le flot d'unités lexicales provenant de l'analyseur lexical contredit les règles grammaticales définissant le langage de programmation.

Ayant une idée précise des erreurs courantes qui peuvent être rencontrées en Pascal, nous avons augmenté la grammaire du langage avec des productions qui engendrent les constructions erronées. Ainsi, si une production d'erreurs est utilisée par l'analyseur syntaxique, celui-ci peut émettre les diagnostics d'erreurs appropriés. Le compilateur indique, alors, grâce à un marqueur de position, la construction erronée qui a été reconnue, et arrête la compilation en ce point.

Au niveau sémantique la détection des erreurs est facilitée par un contrôleur de type. Celui-ci est confondu avec l'analyseur syntaxique pour permettre une compilation en une passe.

A.3.5 - Contrôle de type

Un compilateur doit signaler une erreur si un opérateur est appliqué à un opérande incompatible ; c'est le cas, par exemple, si une variable tableau et une variable fonction sont additionnées.

Celui-ci doit, également, vérifier que le type d'une construction correspond au type attendu par son contexte. Ainsi, par exemple, l'opérateur arithmétique **mod** de Pascal nécessite des opérandes entiers ; le compilateur doit donc vérifier que les opérandes de mod sont bien de type entier. De façon similaire, il doit vérifier que le référençage n'est appliqué qu'à des pointeurs, que seuls les tableaux sont indicés, que toute fonction définie par l'utilisateur est utilisée avec le bon nombre d'arguments de types corrects, ainsi de suite.

On peut avoir besoin d'informations qui ont été calculées antérieurement. Par exemple, des opérateurs comme + peuvent être appliqués soit à des entiers soit à des réels, soit même à d'autres types ; nous devons alors nous intéresser au contexte du + pour déterminer quelle est l'opération voulue.

De plus, certains corps de fonctions ou procédures de la bibliothèque sont polymorphes, c'est-à-dire, qu'ils peuvent être exécutés avec des arguments de plusieurs types. Ils peuvent, de même, renvoyer un résultat dans un type qui diffère de celui de leur argument, ainsi qu'employer, comme toutes fonctions et procédures de Pascal plusieurs paramètres.

A.3.5.1 - Systèmes de typage

La conception d'un contrôleur de type est ainsi fondée sur la connaissance des constructions du langage cible, la notion de type et les règles pour affecter des types aux constructions du langage. Il est implicite qu'un type est associé à chaque expression. En outre, les types ont une structure.

En Pascal comme en C, les types peuvent être soit des types de base soit des types construits. Les types de base sont les types atomiques sans structure interne, ou dont la structure interne n'est pas accessible au programmeur. En Pascal, les types de base sont : booléen, caractère, entier et réel. Les types intervalle et énumération peuvent être traités comme des types de base. Pascal permet au programmeur de construire des types à partir des types de base et d'autres types construits, par exemple les tableaux, des structures et des ensembles. En outre, les pointeurs et les fonctions peuvent également être considérés comme des types construits.

Etant donné que ces différents types sont associés aux variables de l'algorithme par l'intermédiaire de la table des symboles, ceux-ci ne sont qu'un attribut supplémentaire adjoint au symbole considéré. Ainsi, à chacun de ces types nous devons construire un objet approprié qui nous permet d'accéder aux différentes informations qui sont jugées nécessaires pour la compilation.

Si nous prenons par exemple la déclaration d'un type "array" de la forme :

type tableau = **array** [1..20] **of** real;

nous obtenons un attribut associé à la donnée "tableau" qui peut être représenté sous la forme d'un arbre, figure A.8.

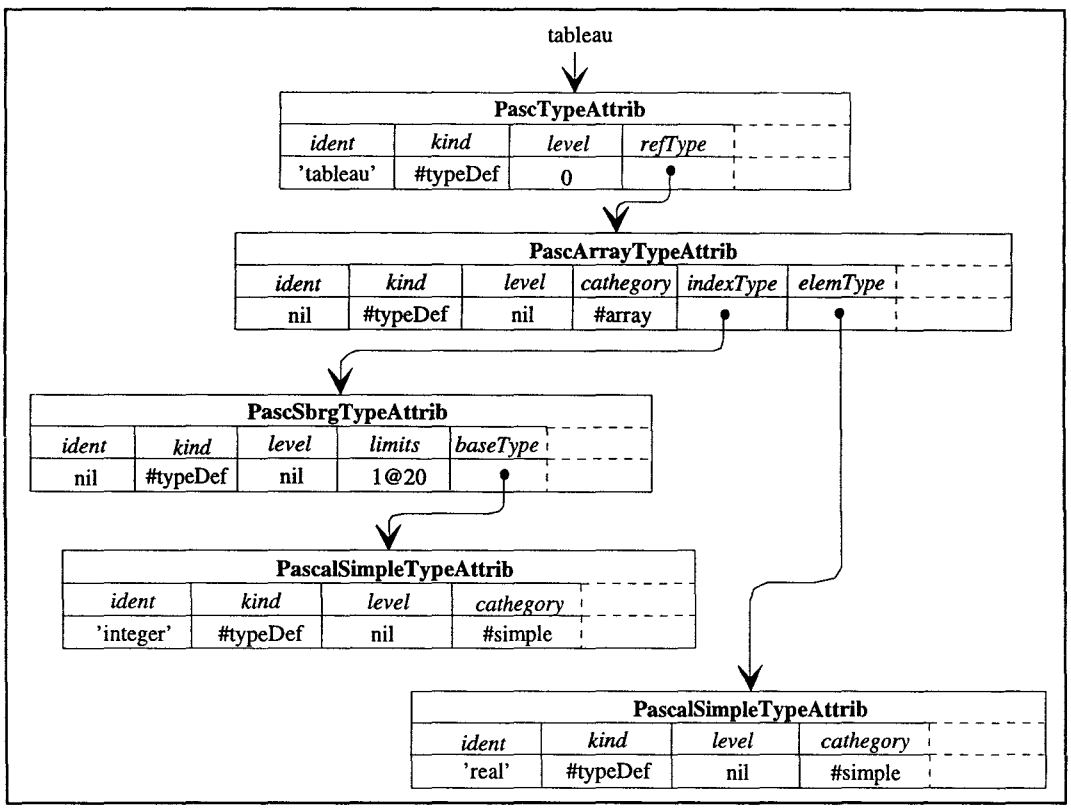


Figure A.8 : Attribut de type "array" présenté sous la forme d'un arbre

Ayant, dans notre compilateur, inséré le contrôleur de type au sein de notre analyseur syntaxique en attachant des attributs aux symboles de la grammaire représentant cette construction, nous avons par conséquent, à chaque symbole de la grammaire, associé un ensemble de règles sémantiques pour calculer la valeur des attributs adjoints aux symboles apparaissant dans cette production. De cette façon, la valeur d'un attribut en un noeud d'un arbre syntaxique est définie par une règle sémantique associée à la production utilisée en ce noeud (cf. figure A.9). L'ordre d'évaluation étant pour nous imposé par la méthode d'analyse, indépendamment des règles sémantiques. Il est bien évident que les valeurs des attributs (synthétisés) des terminaux sont en général fournies par l'analyseur lexical.

Production : E := E1 + E2

Règle sémantique :

```
(E1 attr1 = realAttr | (E1 attr1 = integerAttr))
   & (E2 attr2 = realAttr | (E2 attr2 = integerAttr))
         ifFalse: [^erreur].
E1 attr1 = realAttr or: [E2 attr2 = realAttr]
         ifTrue: [E attr := realAttr]
         ifFalse: [E attr := integerAttr]
```

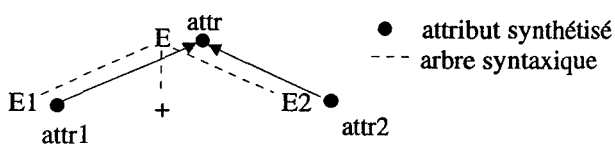


Figure A.9 : Obtention de l'attribut "attr" par la règle sémantique associée à l'arbre syntaxique sous-jacent.

En outre, ayant à résoudre des contraintes liées au polymorphisme, à la surcharge, à la coercition de type, et enfin à l'utilisation d'une bibliothèque externe de fonctions et de procédures, le compilateur doit pouvoir exécuter les différentes règles sémantiques d'une manière récursive et itérative si nécessaire.

Cette vérification est, de même, simplifiée pour la bibliothèque exposé précédemment (cf. Annexe A.2.4.3) en utilisant une table décrivant diverses fonctions et procédures (cf. figure A.10). Celle-ci permet, ainsi, de connaître les types des paramètres d'entrées et de sorties (dans le cas d'une fonction) indépendamment des structures internes qui leur sont propres.

```
libraries put: ((ProcFoncAttrib key: #randomize)
                        resultType: nil).
libraries put: ((ProcFoncAttrib key: #cos)
                        resulType: realAttrib);
                        addParam: #valParam).
```

Figure A.10 : *Introduction dans la table "libraries" de la procédure "randomize" et de la fonction "cos".*

A.4 - PROBLEMES D'INTERACTIVITE DE LA MACHINE VIRTUELLE PASCAL AVEC LE DEBOGUEUR

L'exécution d'un algorithme écrit en Pascal étant visualisée grâce à un objet équivalent au "Débogueur" de Smalltalk, celui-ci nous a imposé quelques contraintes de compilation, pour conserver un ensemble d'interactivités qui lui est propre et lui permettre un fonctionnement optimal.

A.4.1 - Visualisation des variables

Lors de la conception de la machine virtuelle Pascal, nous avons créé une pile chargée de référencer les données de l'algorithme et ainsi d'optimiser le code généré. Par conséquent, nous ne travaillons plus avec les noms de ces données mais avec leurs positions dans la pile (cf. Annexe A.2.3.1).

Ainsi, si le code généré est sémantiquement correct et suffit à interpréter l'algorithme, le débogueur a été conçu pour permettre d'inspecter chacune des variables (globales ou locales) de la méthode traitée. Aussi, celui-ci nous présente à travers un inspecteur les noms des variables visualisables sous la forme d'une liste, nous permettant d'indiquer ou de choisir qu'elle donnée peut-être ou est effectivement visualisée.

En conséquence, il nous faut préalablement enregistrer le nom des variables utilisées par l'algorithme dans une instance quelconque, afin de nous permettre de constituer cette liste. Ceci est réalisé lors de l'initialisation de la pile (de la machine virtuelle Pascal) indépendamment de son fonctionnement originel (cf. Annexe B).

A.4.2 - Initialisation du contexte du débogueur

Lors de la visualisation d'une méthode écrite en Smalltalk et par conséquent éventuellement en Pascal ; le débogueur appelle le compilateur pour effectuer une traduction du texte d'entrée. Ceci lui permet d'initialiser son propre contexte, à savoir entre autres, la mise-à-jour d'une pile lui permettant de gérer chacune des variables temporaires, de classes, d'instances, etc., de la méthode traitée. Cette pile est différente de la pile Pascal car si nous avons un texte d'entrée écrit en Pascal c'est du code Smalltalk qui est interprété. De plus, cette compilation, lui permet d'initialiser une table, en associant le dernier code et la portion de texte à interpréter, déterminant les différentes expressions que l'on veut voir évaluer à chaque pas de la simulation. Cette table est obtenue grâce à l'arbre syntaxique qui, pour chaque noeud, enregistre la partie de texte que l'on veut éventuellement lui faire correspondre. Ceci nous permet de ne plus traiter un programme ligne par ligne comme dans la plupart des débogueur actuels, mais expression par expression. En outre, nous pouvons choisir les expressions qui sont effectivement évaluées visuellement en indiquant ou non, au compilateur, les limites du texte correspondant à l'arbre syntaxique sous-jacent.

Le problème rencontré se situe donc au moment de l'initialisation du contexte du débogueur. Ainsi, lors de la première compilation, possédant le texte source en un seul bloc, il n'est pas difficile de déterminer la portabilité de chacune des procédures, fonctions, paramètres ou variables. Mais, ayant choisi de définir chacun des blocs d'un programme Pascal, comme étant une méthode indépendante, nous avons de même associé à chacune de ces méthodes la partie de texte correspondant au code du bloc considéré tout en englobant le texte des blocs imbriqués. Par conséquent, nous ne compilons que partiellement le texte source pour initialiser le contexte du débogueur. Aussi, nous n'avons plus aucune information sur la portabilité des variables déclarées antérieurement.

Pour pallier ce problème nous avons enregistré, pour chacune des procédures traduites, la table des symboles sous-jacente. Ainsi, le compilateur, peut mettre à jour une nouvelle table des symboles lors de l'initialisation du contexte du débogueur, et de même, connaître les attributs des différentes données précédant la procédure que nous voulons traduire.

A.4.3 - Compilation d'expressions

Ne nous satisfaisant pas de visualiser les différentes variables d'un algorithme nous avons donné la possibilité d'inspecter chacune des expressions que l'utilisateur juge utile à sa compréhension. Mais il faut, pour ce faire, pouvoir compiler une simple partie de texte en lui signifiant qu'il fait partie du contexte de la méthode traitée par le débogueur. De même, nous sommes amenés, si celui-ci fait partie d'une procédure, à initialiser sa table des symboles comme précédemment.

Il est entendu, que le code généré n'est valable que pour le contexte dans lequel il a été compilé. Ainsi, pour un autre contexte il faut soit lui signifier simplement qu'il a changé de contexte (lors d'appels de méthodes récursives) soit qu'il faut recompiler cette expression avec le nouveau contexte (le code intermédiaire précédemment engendré ne correspondant plus aux mêmes données).

A.5 - CONCLUSION

Nous avons ainsi déterminé un modèle de machine virtuelle Pascal nous permettant de compiler un algorithme écrit en Pascal sous Smalltalk. Bien qu'il ne puisse, jusqu'à ce jour, compiler toutes les fonctions et toutes les formes évoluées de la grammaire Pascal ; nous avons un modèle suffisamment ouvert pour nous permettre d'enrichir celui-ci sans remettre en cause les principes de bases énoncés précédemment.

Ce compilateur va nous permettre de nous détacher des contraintes de la syntaxe Smalltalk et ainsi rendre notre application utilisable par un plus large public. Celui-ci pouvant alors se composer d'étudiants apprenant un langage structuré tel que le Pascal et de même apporter une pédagogie nouvelle en visualisant les données de l'algorithme qu'ils auront écrit sous la forme d'images animées. De même, ils pourront utiliser un langage orienté objet tel que Smalltalk puisque suivant la syntaxe employée le compilateur effectue la traduction du texte source automatiquement dans le langage employé.

Dans l'annexe B nous présentons la traduction d'un algorithme pascal suivant les deux modèles de machine virtuelle que nous avons conçue.

ANNEXE B

TRADUCTION D'UN PROGRAMME PASCAL SOUS SMALLTALK

```
program rapidtripas;

const nbrelem = 20;

type tableau = Array [1..nbrelem] of real ;

var  k: integer;
     tab: tableau;


procedure rapidtri (var aTab: tableau; bas, haut: integer);
var b, h: integer;

procedure echange;
var tempo : real;

begin
          tempo := aTab[h];
          aTab[h] := aTab[b];
          aTab[b] := tempo
end;


begin
          b := bas;
          h := haut;
          repeat
                    while (aTab[b] <= aTab[h]) and (b < h) do h :=h - 1;
                    echange;
                    while aTab[b] < aTab[h] do b :=b + 1;
                    echange;
          until b = h;
          if bas < (h - 1) then  rapidtri (aTab, bas,h - 1);
          if (b + 1) < haut then rapidtri (aTab, b + 1, haut);
end;

begin
          randomize;
          for k := 1 to nbrelem do tab[k] := random(100);
          rapidtri (tab, 1, nbrelem);
end.
```

Figure B.1 : *Exemple d'un algorithme écrit dans la syntaxe Pascal appelé communément "le Tri shell"*

```
!Rapidtripas methodsFor: 'exemple'!

rapidtripas
                    | t1 t2 t3 t4 t5 |
                    t1 := 20.
                    t4 := (t2 := ArraySim new: t1 - 1 + 1) deepCopy.
                    AlgoDebuger openAlgo.
                    t3 := 1.
                    [t3 <= t1] whileTrue: [  t4 at: t3 put: Random new next * 100.
                                             t3 := t3 + 1].
                    t5 := ((((Array with: t4) with: t1) with: t2) with: t3) with: t4.
                    self rapidtri: 1 with: t1 with: t5.
                    t4 := t5 at: 1.
                    t1 := t5 at: 2.
                    t2 := t5 at: 3.
                    t3 := t5 at: 4.
                    t4 := t5 at: 5.
                    ^self

rapidtri: t1 with: t2 with: t3
                    | t4 t5 t6 |
                    t4 := t1.
                    t5 := t2.
                    [[((t3 at: 1) at: t4) <= ((t3 at: 1) at: t5) & (t4 < t5)]
                            whileTrue: [t5 := t5 - 1].
                    t6 := ((((((Array with: t4) with: t5) with: (t3 at: 1)) with: (t3 at: 2)) with: (t3 at: 3))
                                        with: (t3 at: 4)) with: (t3 at: 5).
                    self echange: t1 with: t2 with: t6.
                    t4 := t6 at: 1.
                    t5 := t6 at: 2.
                    t3 at: 1 put: (t6 at: 3).
                    t3 at: 2 put: (t6 at: 4).
                    t3 at: 3 put: (t6 at: 5).
                    t3 at: 4 put: (t6 at: 6).
                    t3 at: 5 put: (t6 at: 7).
                    [((t3 at: 1) at: t4) < ((t3 at: 1) at: t5)]
                            whileTrue: [t4 := t4 + 1].
                    t6 := ((((((Array with: t4) with: t5) with: (t3 at: 1)) with: (t3 at: 2)) with: (t3 at: 3))
                                        with: (t3 at: 4)) with: (t3 at: 5).
                    self echange: t1 with: t2 with: t6.
                    t4 := t6 at: 1.
                    t5 := t6 at: 2.
                    t3 at: 1 put: (t6 at: 3).
                    t3 at: 2 put: (t6 at: 4).
                    t3 at: 3 put: (t6 at: 5).
                    t3 at: 4 put: (t6 at: 6).
                    t3 at: 5 put: (t6 at: 7).
                    t4 = t5] whileFalse.
                    t1 < (t5 - 1) ifTrue: [t6 := ((((Array with: (t3 at: 1)) with: (t3 at: 2)) with: (t3 at: 3))
                                                        with: (t3 at: 4)) with: (t3 at: 5).
                                        self rapidtri: t1 with: t5 - 1 with: t6.
                                        t3 at: 1 put: (t6 at: 1).
                                        t3 at: 2 put: (t6 at: 2).
                                        t3 at: 3 put: (t6 at: 3).
                                        t3 at: 4 put: (t6 at: 4).
                                        t3 at: 5 put: (t6 at: 5)].
                    t4 + 1 < t2 ifTrue: [t6 := ((((Array with: (t3 at: 1)) with: (t3 at: 2)) with: (t3 at: 3))
                                                        with: (t3 at: 4)) with: (t3 at: 5).
                                        self rapidtri: t4 + 1 with: t2 with: t6.
                                        t3 at: 1 put: (t6 at: 1).
                                        t3 at: 2 put: (t6 at: 2).
                                        t3 at: 3 put: (t6 at: 3).
                                        t3 at: 4 put: (t6 at: 4).
                                        t3 at: 5 put: (t6 at: 5)].
                    ^self


echange: t1 with: t2 with: t3

                | t4 |
                t4 := (t3 at: 3) at: (t3 at: 2).
                (t3 at: 3) at: (t3 at: 2) put: ((t3 at: 3) at: (t3 at: 1)).
                (t3 at: 3) at: (t3 at: 1) put: t4.
                ^self  ! !
```

Partie déclarative

Ajoue de code pour récupérer les valeurs des variables

Figure B.2 : *Obtention de trois méthodes Smalltalk*
après traduction du programme Pascal de la figure B.1, suivant le premier modèle

```
!Rapidtripas methodsFor: 'internal procedures'!

init.typePool
        (typePool := Array new: 6) at: 1 put: PascVInteger new; at: 2 put: PascVBoolean new; at: 3 put: PascVChar new; at: 4 put: PascVReal new;
        at: 5 put: PascVString new; at: 6 put: (PascVArray new put: ((ArraySim new: 20)
                                                collect: [:t1 | (typePool at: 4) valueCopy])).
        ^self
rapidtripas
        | t1 |
        (t1 := PascalSF frameFor: ((Array new: 2)
                                at: 2 put: 'tab';yourself;
                                at: 1 put: 'k';yourself)
                pars: (Array new: 0)
                args: (Array new: 0)
                linkTo: nil) vrb: 2 put: (typePool at: 6) valueCopy.
        t1 vrb: 1 put: (typePool at: 1) valueCopy.
        AlgoDebuger openAlgo.
        (procPool at: #stdProcFun.randomize) valueWithArguments: (Array new: 0).
        (t1 down: 0 vrb: 1) put: (PascVInteger new put: 1).
        [(t1 down: 0 vrb: 1) vValue <= (PascVInteger new put: 20) vValue]
                whileTrue: [((t1 down: 0 vrb: 2) at: (t1 down: 0 vrb: 1) vValue - 0)
                                put: ((procPool at: #stdProcFun.random)
                                        valueWithArguments: ((Array new: 1)
                                                                at: 1 put: (PascVInteger new put: 100) valueCopy; yourself)).
                        (t1 down: 0 vrb: 1) put: (t1 down: 0 vrb: 1) vValue + 1].
        self rapidtri: (t1 down: 0) args: ((Array new: 3)
                                        at: 1 put: (t1 down: 0 vrb: 2); yourself;
                                        at: 2 put: (PascVInteger new put: 1) valueCopy; yourself;
                                        at: 3 put: (PascVInteger new put: 20) valueCopy; yourself).
        ^self.

rapidtri: t1 args: t2
        | t3 |
        (t3 := PascalSF frameFor: ((Array new: 2)
                                at: 1 put: 'b';yourself;
                                at: 2 put: 'h';yourself)
                pars: ((Array new: 3)
                        at: 3 put: 'haut';yourself;
                        at: 2 put: 'bas';yourself;
                        at: 1 put: 'atab';yourself)
                args: t2
                linkTo: t1) vrb: 1 put: (typePool at: 1) valueCopy.
        t3 vrb: 2 put: (typePool at: 1) valueCopy.
        (t3 down: 0 vrb: 1) put: (t3 down: 0 prm: 2).
        (t3 down: 0 vrb: 2) put: (t3 down: 0 prm: 3).
        [[(PascVBoolean new put: (((t3 down: 0 prm: 1) at: (t3 down: 0 vrb: 1) vValue - 0)
                <= ((t3 down: 0 prm: 1) at: (t3 down: 0 vrb: 2) vValue - 0)) vValue
                        & (PascVBoolean new put: (t3 down: 0 vrb: 1) < (t3 down: 0 vrb: 2))) vValue]
                        whileTrue: [(t3 down: 0 vrb: 2) put: (t3 down: 0 vrb: 2) - (PascVInteger new put: 1)].
        self echange: (t3 down: 0) args: (Array new: 0).
        [(((t3 down: 0 prm: 1) at: (t3 down: 0 vrb: 1) vValue - 0) < ((t3 down: 0 prm: 1) at: (t3 down: 0 vrb: 2) vValue - 0)) vValue]
                        whileTrue: [(t3 down: 0 vrb: 1) put: (t3 down: 0 vrb: 1) + (PascVInteger new put: 1)].
        self echange: (t3 down: 0) args: (Array new: 0).
        ((t3 down: 0 vrb: 1) = (t3 down: 0 vrb: 2)) vValue] whileFalse.
        ((t3 down: 0 prm: 2) < ((t3 down: 0 vrb: 2) - (PascVInteger new put: 1))) vValue
                        ifTrue: [self rapidtri: (t3 down: 1) args: ((Array new: 3)
                                                                at: 1 put: (t3 down: 0 prm: 1); yourself;
                                                                at: 2 put: (t3 down: 0 prm: 2) valueCopy; yourself;
                                                                at: 3 put: (t3 down: 0 vrb: 2) - (PascVInteger new put: 1); yourself)].
        ((t3 down: 0 vrb: 1) + (PascVInteger new put: 1) < (t3 down: 0 prm: 3)) vValue
                        ifTrue: [self rapidtri: (t3 down: 1) args: ((Array new: 3)
                                                                at: 1 put: (t3 down: 0 prm: 1); yourself;
                                                                at: 2 put: (t3 down: 0 vrb: 1) + (PascVInteger new put: 1); yourself;
                                                                at: 3 put: (t3 down: 0 prm: 3) valueCopy; yourself)].
        ^self

echange: t1 args: t2
        | t3 |
        (t3 := PascalSF frameFor: ((Array new: 1)
                                at: 1 put: 'tempo'yourself)
                pars: (Array new: 0)
                args: t2
                linkTo: t1) vrb: 1 put: (typePool at: 4) valueCopy.
        (t3 down: 0 vrb: 1) put: ((t3 down: 1 prm: 1) at: (t3 down: 1 vrb: 2) vValue - 0).
        ((t3 down: 1 prm: 1) at: (t3 down: 1 vrb: 2) vValue - 0) put: ((t3 down: 1 prm: 1) at: (t3 down: 1 vrb: 1) vValue - 0).
        ((t3 down: 1 prm: 1) at: (t3 down: 1 vrb: 1) vValue - 0) put: (t3 down: 0 vrb: 1).
        ^self ! !
```

initialisation de la pile

***Figure B.3 :** Obtention de quatre méthodes Smalltalk*
après traduction du programme Pascal de la figure B.1, suivant le second modèle

ANNEXE C

LES MODÈLES MULTIAGENT, M.V.C., M.V.C. ETENDU ET P.A.C.

C.1 - LES MODELES MULTIAGENT

On a constaté, durant ces dernières années, une évolution considérable des interfaces, accompagnée d'un changement radical de la philosophie d'interaction [MOS 94a].

En effet, avec la nouvelle génération d'interface on passe d'un ancien type de dialogue contrôlé par l'ordinateur, qui présente à l'utilisateur des écrans successifs, à un type de dialogue contrôlé par les actions de l'utilisateur, traduites sous forme d'événements auxquels l'ordinateur doit répondre.

Une des difficultés majeures dans la construction des interfaces utilisateurs réside dans le contrôle de ces actions et la gestion du dialogue d'une manière générale. Avec les modèles précédents, on n'avait pas de stratégies claires et faciles à mettre en œuvre pour le contrôle de dialogue. Avec le modèle multiagent, l'utilisation du paradigme objet nous amène à répartir le contrôle sur une multitude d'objets de présentation appelés aussi objets interactifs.

Ces objets interactifs (fenêtres, icônes, processus, boutons, menus, etc.) manipulés dans l'interface, apparaissent comme des médiateurs entre le monde abstrait du système et le monde concret de l'utilisateur.

Les événements, actions de l'utilisateur sur les objets, se traduisent en messages adressés à ces objets. Ces messages sélectionnent les méthodes responsables des comportements internes et externes des objets.

Son principe de répartition du dialogue lui confère la possibilité de gestion de la composition et l'interaction de la multitude d'objets mis en œuvre dans le dialogue.

L'utilisation du paradigme objet et de sa forte modularité, au niveau de l'abstraction, permet une conception itérative des interfaces utilisateurs.

Plusieurs modèles ont été proposés et s'appuient tous sur le paradigme objet (agent) et ont tous, comme point commun, la distinction entre l'abstraction appartenant à l'application et la présentation de l'interface. Ces principaux modèles sont : le modèle M.V.C. (Modèle, Vue, Contrôleur) de Smalltalk, le modèle M.V.C. étendu de Ralph L. London et Robert A. Duisberg [LON 85] qui essaye de palier les défauts du modèle M.V.C. et le modèle P.A.C. de Joëlle Coutaz [COU 90b].

C.2 - Le Modèle M.V.C. . (Modèle, Vue, Contrôleur)

Le modèle M.V.C. (Cf. chapitre III.3) reste, à l'heure actuelle, l'une des meilleures solutions pour la construction rapide d'interfaces interactives, en ayant un minimum de pratique et d'expérience de la bibliothèque de classe du langage Smalltalk.

Pour bâtir une animation à partir de ce modèle, il faut :

1) Définir les différents objets dont on veut animer le comportement (ensemble des objets {O}) appartenant à l'objet "Modèle".

2) Définir leur(s) représentation(s) graphique(s), constituant la/les "Vue(s)".

3) Mettre en relation chaque objet de l'ensemble {O} avec sa représentation graphique de telle sorte que celle-ci reflète toujours, très exactement, l'état de l'objet auquel elle est associée (notion de dépendance).

4) Associer éventuellement à la paire "objet-représentation ou vue" un "Contrôleur" permettant à l'utilisateur d'intervenir sur le comportement des objets représentés au moyen de la souris ou du clavier.

La structure du modèle M.V.C. semble ainsi effectivement adaptée à la conception d'animations graphiques interactives mais présente un inconvénient majeur.

C.2.1 Inconvénient majeur du modèle M.V.C.

Bâtir une animation sur le système M.V.C. (Cf. paragraphe III.3) présente un inconvénient majeur car sa structure implique que les différentes séquences graphiques, illustrant les actions sur le modèle, soient introduites dans la définition même des actions.

Par exemple pour construire une animation sur le comportement d'une pile, il faut fournir une représentation graphique de la pile et associer une séquence d'événements graphiques à l'arrivée d'une donnée dans la pile ou à son retrait. Aussi, tout le code graphique relatif au déplacement d'une donnée dans la pile est inséré dans une méthode appelée par exemple 'empiler: uneDonnée' définie pour la classe "Pile".

En d'autres termes, l'application et l'animation sont indissociables : l'application ne peut pas tourner sans l'animation et l'animation ne peut pas être transposée sur une application.

C.3 - LE MODELE M.V.C. ETENDU

Pour résoudre ce problème de "portabilité" des animations, Ralph L. London et Robert A. Duisberg ont étendu la structure M.V.C. en y ajoutant une quatrième entité, la routine graphique.

Les relations entre le modèle, la vue, le contrôleur et la routine graphique associés, sont représentées par la figure C.1.

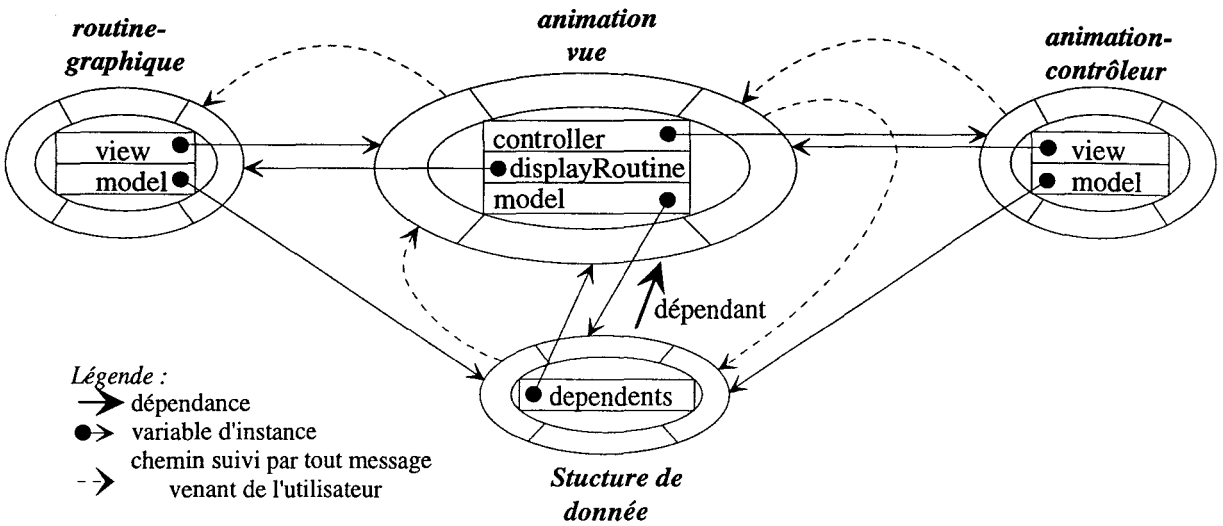


***Figure C.1** : Relations du modèle M.V.C. étendu*

L'objet routine-graphique regroupe toutes les méthodes spécifiques à la création et à la gestion d'une animation reflétant l'état et le comportement d'un objet.

Dans cette nouvelle structure, désignée comme structure d'animation, R. London et R. Duisberg ont généralisé le rôle du contrôleur et de la vue en créant deux nouvelles classes, "AnimationView" et "AnimationController", sous-classes respectives de "StandardSystemView" et "MouseMenuController". Quelque soit la structure de donnée, c'est-à-dire quelque soit le modèle dont on désire animer le comportement, la vue et le contrôleur qui lui seront associés, resteront toujours des instances de "AnimationView" et de "AnimationController". Le rôle de ces deux objets se borne, maintenant, à celui de relais. Le contrôleur sert de relais entre l'utilisateur et le modèle (le receveur gérant le message du menu n'est plus le contrôleur lui même mais son modèle), la vue sert de relais entre le modèle et la routine graphique associée (la demande de mise à jour transite bien, par la relation de dépendance, vers la vue associée au modèle mais elle est aussitôt transmise à la routine graphique qui réalise la mise à jour effective). Le message de mise à jour, 'update: unEvènementIntéressant', a pour cette raison été défini dans la classe "AnimationView".

C.3.1 - Exemple d'application du modèle M.V.C. étendu

L'exemple présenté ci-dessous est le cas d'un système "Producteur/Consommateur". Celui-ci met en relation quatre types d'objets : un producteur, un consommateur, un magasin, et un moniteur gérant les accès au magasin. Les relations liant ces différents objets peuvent être modélisées comme à la figure C.2.

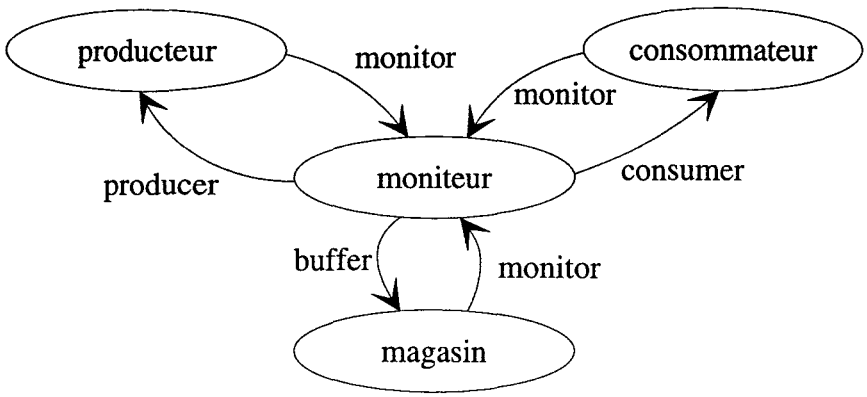


***Figure C.2 :** Relations liant les objets d'un "Producteur/Consommateur"*

Dans cette application, l'utilisateur commande à son gré la production ou la consommation. Dans le cas d'une production, une donnée transite du producteur vers le magasin. Elle transitera du magasin vers le consommateur dans le cas d'une consommation. Si le magasin est vide, l'utilisateur ne peut consommer. Sa demande doit rester en attente jusqu'à ce qu'une donnée ait été introduite. Inversement, si le magasin est plein, l'utilisateur ne peut plus produire. Sa demande doit rester en attente jusqu'à ce qu'une donnée ait été consommée.

Une représentation graphique de l'ensemble est donnée par la figure C.3.

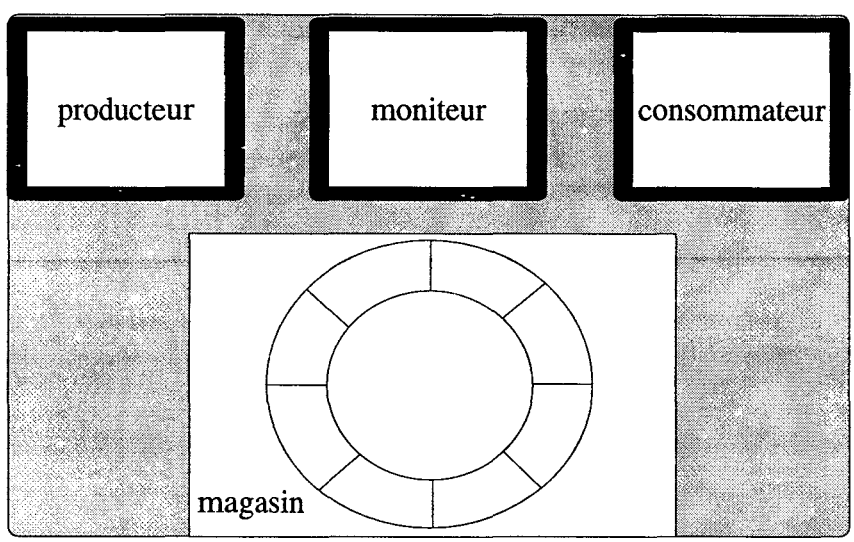


***Figure C.3 :** Représentation graphique d'un "Producteur/Consommateur"*

La structure d'animation associée à cette application est représentée par la figure C.4.

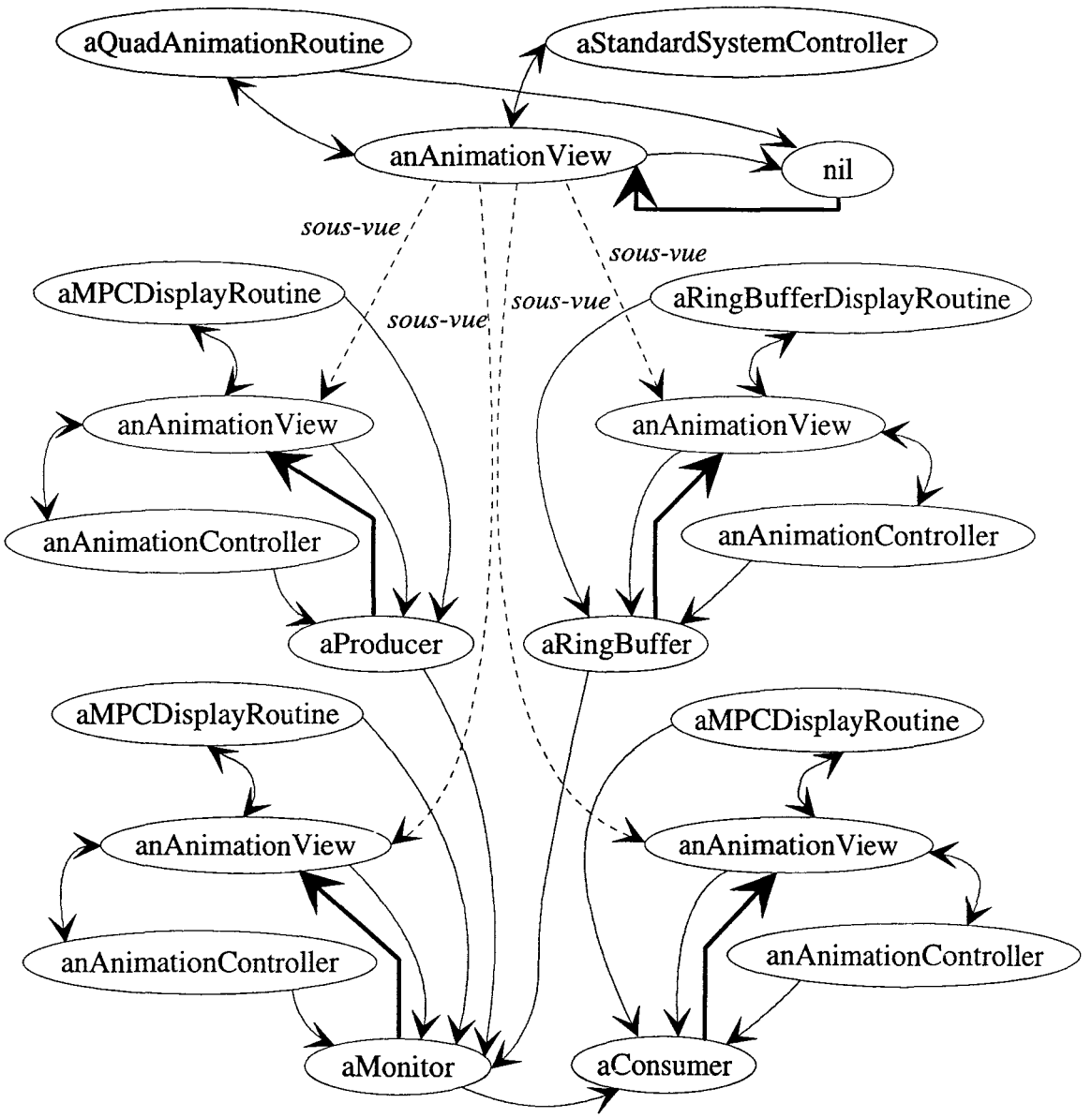


Figure C.4 : *Structure d'un "Producteur/consommateur" pour le modèle M.V.C. étendu*

Si le modèle M.V.C. étendu permet de résoudre le problème de portabilité d'une animation il engendre et conserve d'autres inconvénients du modèle M.V.C. de Smalltalk.

C.3.2 - Inconvénients du modèle M.V.C. étendu

L'inconvénient inhérent au modèle M.V.C. étendu est l'obtention d'une structure d'animation complexe. Par exemple la structure de la figure C.4 ne met en relation que quatre types d'objets principaux, ce qui laisse présager des difficultés non seulement de définition mais surtout de maintenance des applications bâties sur ce modèle.

Par ailleurs, les interfaces qui utilisent le principe M.V.C. restent en grande partie procédurale. Ceci entraîne nécessairement des échanges par messages qui risquent d'allonger

le temps de réaction de l'animation. De plus, dans une structure M.V.C. on distingue l'abstraction, représentée par le "Modèle", de la représentation représentée par la "Vue". Le passage de l'abstraction à l'affichage sur écran est une opération assurée par un certain nombre de méthodes, réparties entre le modèle et la vue. L'inconvénient est que le modèle M.V.C. n'offre pas de stratégie claire de répartition de ces méthodes.

C.4 - LE MODELE P.A.C. (PRESENTATION, ABSTRACTION, CONTROLEUR)

Le modèle P.A.C. structure l'architecture d'un système interactif en une multitude d'agents (ou objets interactifs), organisés en trois classes de compétence (Cf. figure C.4).

- La présentation définit l'image de l'agent, c'est-à-dire son comportement perceptible à l'utilisateur.

- L'abstraction définit les fonctions et les attributs internes de l'agent, c'est-à-dire son comportement vis-à-vis d'autres agents,

- Le contrôle maintient la cohérence entre les deux perspectives.

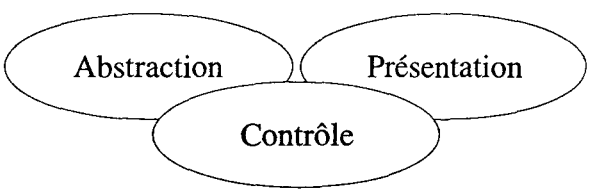


Figure C.5 : Structure du modèle P.A.C.

Un système interactif est vu comme une hiérarchie d'objets P.A.C., allant du niveau le plus haut au niveau le plus bas (Cf. figure C.5).

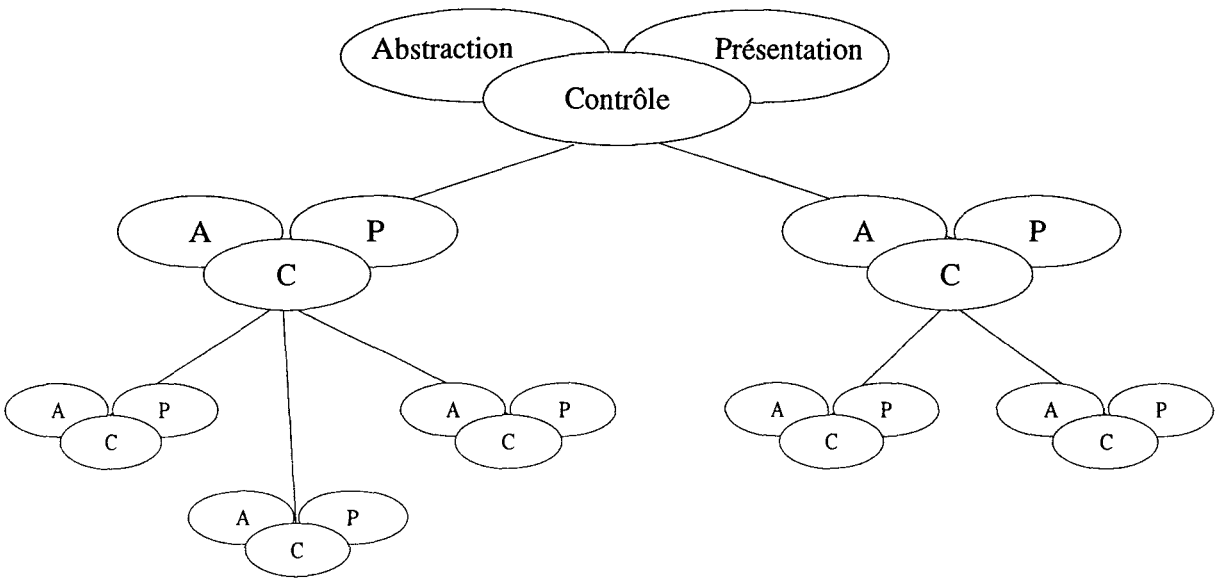


Figure C.6 : *Vue d'un système interactif*

Du point de vue du contrôle du niveau le plus haut, l'application est un "producteur / consommateur" de valeurs qui modélisent les concepts du domaine. Une des fonctions du contrôle du niveau le plus haut est de mettre en correspondance les concepts de l'application avec les abstractions des objets interactifs du niveau le plus bas. Un objet interactif peut être élémentaire ou composé de :

- un objet interactif élémentaire est une unité de compétence et d'exécution répartie en trois classes de fonctions : Présentation, Abstraction et contrôle,

- un objet interactif composé définit une nouvelle unité de compétence, organisée selon les mêmes critères qu'un objet élémentaire, mais qui en plus, hérite du comportement des objets constituant et les soumet à ces propres règles.

C.4.1 - Exemple d'application du modèle P.A.C.

L'exemple présenté ci-dessous est le système "Thermo" réalisé en "C" dans un environnement mono tâche [COU 90b] qui a pour objectif de présenter les relations entre les notions de chaleur, de température et de volume d'eau.

Un thermomètre indique la température actuelle, un réchaud fait office de source de chaleur, un récipient sert à recevoir un certain volume de liquide et un distributeur permet d'obtenir de l'eau.

La figure C.6 décrit l'architecture P.AC du système "Thermo". L'abstraction du niveau le plus haut modélise les lois de la thermodynamique. Elle est capable de déterminer, à tout instant, la chaleur produite par une résistance, de calculer la température d'un volume d'eau et de déterminer si cette eau est en état d'ébullition ou non. La présentation du niveau le plus haut, n'a pas d'attributs particuliers, elle est en faite considérée comme inexistante.

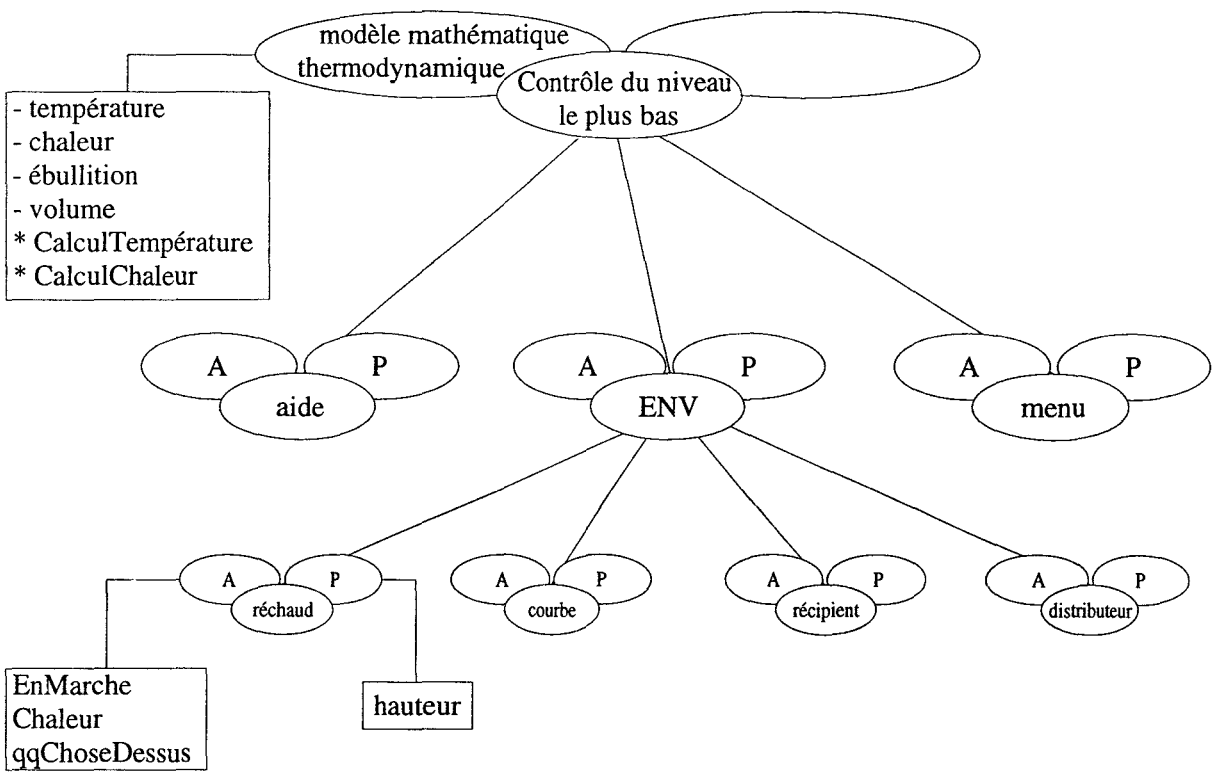


Figure C.7 : Architecture P.A.C. du système "Thermo"

Les objets composants se répartissent en deux catégories : ceux qui reproduisent des objets du monde réel, en rapport étroit avec les fonctions du système telles que le "Réchaud", le "Récipient" et le "Distributeur", et ceux d'utilité générale tels que l'aide, les boutons et les menus.

C.4.2 - Echanges entre l'application et l'interface

Dans le cas du système "Thermo", l'application parle de température mais doit tout ignorer des techniques de présentation. De même, le thermomètre du récipient peut servir à présenter d'autres valeurs que celles du système "Thermo". L'application et le récipient ne communiquent jamais directement mais passent par une indirection gérée dans le niveau le plus haut.

Dans le cas du système "Thermo", la technique des tables de correspondance a été choisie pour réaliser les indirections dont les liaisons les plus significatives sont représentées par la figure C.7.

Dans le sens application vers interface :

- l'entier "température" qui modélise la température est lié à l'abstraction "Temp" d'un objet de classe "Récipient",
- l'entier "chaleur" est lié à l'abstraction "Chaleur" d'un objet de la classe "Réchaud".

Dans le sens interface vers application :

- l'abstraction "En Marche" d'un objet de la classe "Réchaud" est liée à la fonction "CalculChaleur" de l'application,
- l'abstraction "qqChoseDessus" d'un objet de la classe "Réchaud" est liée à la fonction "CalculTempérature" de l'application.

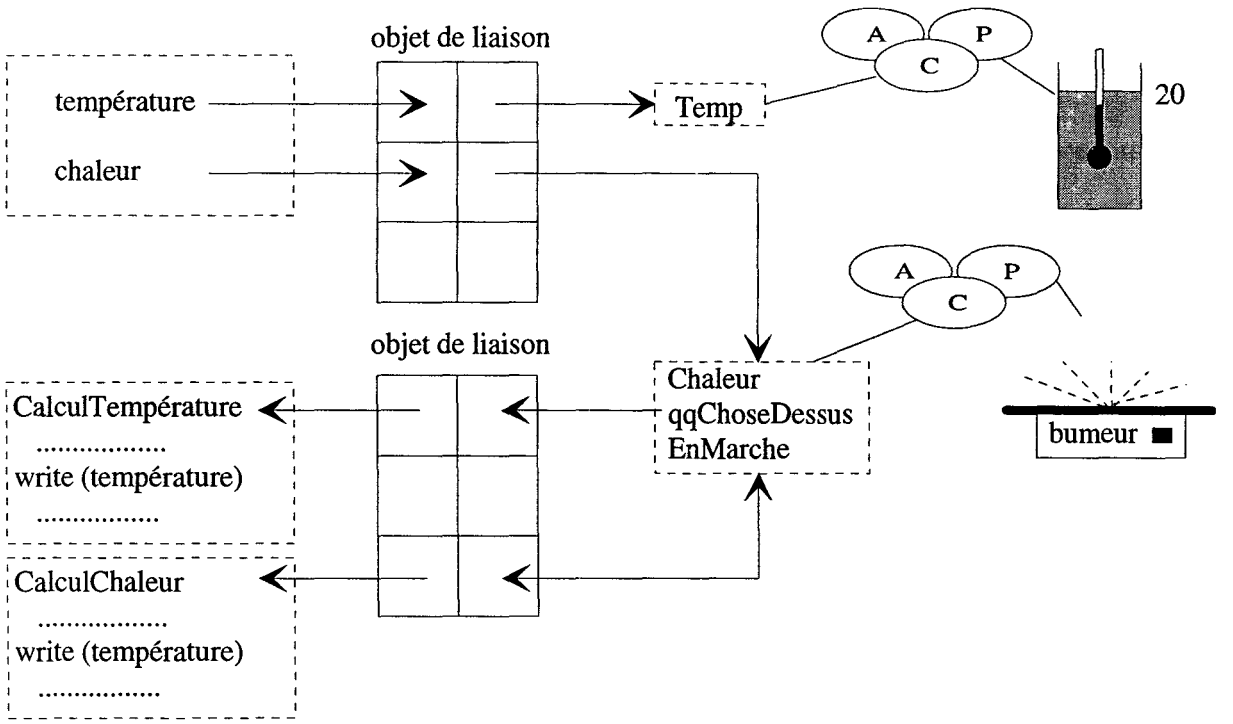


Figure C.8 : *Echanges entre l'application et l'interface*

Lorsque l'application modifie un concept, le contrôle exprime le changement dans les termes des abstractions des objets qui lui sont associées. Les contrôles de ces objets prennent le relais jusqu'à ce que l'effet se traduise dans les présentations des objets élémentaires. Dans le sens inverse, les actions de l'utilisateur sont interprétées par les présentations de plus bas niveau, puis colportées en remontant la hiérarchie jusqu'à ce que des abstractions d'objets interactifs soient en liaison directe avec les concepts de l'application.

Le modèle P.A.C. a l'avantage de s'appuyer sur les principes directeurs du modèle multiagent, ce qui lui confère une organisation modulaire et une ouverture vers les traitements physiquement distribués mais présente également des inconvénients.

C.4.3 - Inconvénients du modèle P.A.C.

Comme il a été précédemment décrit dans l'exemple "Thermo", l'architecture P.A.C. d'un système interactif est représentée par une structure hiérarchique. Au sommet de la hiérarchie se trouve l'abstraction et le contrôle du niveau le plus haut. L'abstraction constitue le modèle mathématique de l'application concernée. Celle du système "Thermo", modélise la loi de la thermodynamique.

En fait, le principe de répartition sur une multitude d'agents n'est appliqué que sur la présentation graphique. Ceci limite l'application du principe métaphorique du monde réel qui considère les objets interactifs comme des objets complets ayant un comportement, interne pour décrire les lois physiques de l'objet représenté, et externe pour la communication avec l'environnement.

Cette limitation engendre un inconvénient : le risque d'allonger le temps de réaction du système. En effet si un objet subit une modification, celle-ci est d'abord interprétée par sa représentation puis reflétée dans l'abstraction du même niveau et enfin colportée en remontant la hiérarchie jusqu'au niveau le plus haut, ou se décidera les réactions nécessaires pour une éventuelle mise à jour.

D'autre part, le modèle P.A.C. introduit la notion de contrôle ayant la charge de faire cohabiter deux environnements totalement indépendants, en appliquant le principe d'indirection. Par exemple, dans le sens application => interface, toute modification d'abstraction est signalée au contrôle et traduit l'effet sur l'ensemble des présentations liées à cette abstraction. Cependant le modèle P.A.C. ne précise pas comment se fait cette traduction ni comment on peut créer et gérer l'animation.

BIBLIOGRAPHIE

[ABO 85] **Above Systems, Inc.** (1985). "*Postscript Language Reference Manual*" Addison-Wesley, Menlo.

[AHO 86] **A. V. Aho, R. Sethi, et J. D. Ullman** (1986). "*Compilers: Principles, Techniques, and Tools*" Addison-Wesley, Reading, Mass.

[AMB 89] **A. L. Ambler et M. M. Burnett** (October 1989). "*Influence of visual technology on the evolution of language environments*" Computer, Vol. 22, N°10, pp 9-22.

[AMB 90] **A. L. Ambler et M. M. Burnett** (1990). "*Visual Forms of Iteration that Preserve Single Assignment*", Journal of Visual Languages and Computing, Vol. 1, N°2, Academic Press.

[AMB 93] **A. L. Ambler et Y. T. Hsia** (1993). "*Generalizing Selection in By Demonstration Programming*". Journal of Visual Language and Computing, Vol. 4, N°3, Sept., pp. 9-22.

[AMM 77] **U. Ammann** (1971). "*On Code Generation in a Pascal Compiler*" Software - Practice and Experience, Vol. 7, N°3, pp. 391-423.

[AMM 81] **U. Ammann** (1981). "*The Zürich Implementation*" In Barron, pp. 63-82.

[APP 85] **Apple Computer, Inc.** (1985). Inside Macintosh, Addison-Wesley, CA.

[ASE 87] **P. J. Asente** (1987). "*Editing Graphical Objects Using Procedural Representations*". PhD thesis, Computer Systems Laboratory, Standford University, Palo Alto, CA.

[BAE 68] **R. M. Baecker** (1968). "*Experiments in On-Line Graphical Debugging: The Interrogation of Complex Data Structures (Summary Only)*". In Proceedings of First Hawaii International Conference on the System Sciences, January, pp. 128-129.

[BAE 69] **R. M. Baecker** (1969) "*Interactive computer mediated Animation*". PhD. O. Dissertation, MIT, Project Mac-TrG1.

[BAE 75] **R. M. Baecker** (1975). "*Two systems which produce animated representations of the execution of computer programs*". ACM SIGCSE Bulletin N°7, pp. 158-167.

[BAE 81] **R. M. Baecker** (1981). "*Sorting Out Sorting*". Morgan Kaufmann, Los Altos, California. Narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81 and excepted in ACM SIGGRAPH Video Review N°7, 1983.

[BAE 90] **R. M. Baecker et A. Marcus** (1990). "*Human factors and Typography for More Readable Programs*". Addison-Wesley, Reading, Massachusetts.

[BAL 96] **T. Ball et S. G. Eick** (1996). "*Software Visualization in the large*". IEEE Computer, N°4, pp. 33-43

[BAS 85] **D. B. Baskerville** (1985). "*Graphic presentation of data structures in the DBX debugger*". Technical Report UCB/CSD 86/260, University of California at Berkeley, CA, October.

[BEA 88] **M. Beaudouin-Lafon et S. Karsenty** (1988). "*Prototyping user interfaces for applications depicted by graphs*". In Proceedings of the 21st Hawaï International Conference on System Sciences, Kailua-Kona, HI, pp. 436-445.

[BEL 87] **P. C. Bell et R. M. O'Keefe** (1987) "*Visual interactive Simulation : History, recent developments and major issues*". Simulation, Vol. 49, N°3, pp 1-6.

[BEN 91] **J. L. Bentley et B. W. Kernighan** (1991). "*A System for Algorithm Animation*". Computing Systems, Vol. 4, N°1, pp. 5-30.

[BEN 92] **J. L. Bentley et B. W. Kernighan** (1992). "*ANIM*". Murray Hill, NJ: AT&T Bell Laboratories. a collection of ANSI C programs used to visualize programs, runs on any UNIX computer. available by anonymous ftp from research.att.com in /netlib/research.

[BEP 80] **L. A. Belady, C. J. Evangelista et L. R. Power** (1980). "*GREENPRINT: A graphics representation of structured programs*". IBM Systems Journal, Vol. 19, N°4, pp. 79-90.

[BOC 86] **H. D. Bocker, G. Ficher et H. Nieper** (1986). "*The enhancement of understanding through visual representations*". In Proceeding of the ACM SIGCHI'86 Conference on Human factors in Computing Systems, Boston, MA, April 1986, pp. 44-50.

[BOO 91] **G. Booch** (1991). "*Object-Oriented Design with applications*". Benjamin / Cummings, New York.

[BOR 79] **A. Borning** (1979). "*Thinglab - A Constraint-Oriented Simulation Laboratory*". PhD thesis, Standford University, Palo Alto, CA, March.

[BOR 81] **A. Borning** (1981). "*The programming language aspects of Thinglab*". ACM TOPLAS, Vol. 3, N°4, pp. 353-487.

[BRA 87] **G. Brassard et P. Bratley** (1987). "*Algorithmique : conception et analyse*". Manuels Informatiques, Ed. Masson, Les presses de l'Université de Montréal.

[BRA 90] **M. Brayshaw** (1990). "*Visual Models of PARLOG Execution*" (Technical Report No. 64). Human Cognition Research Laboratory, The Open University, Milton Keynes, England.

[BRA 91a] **M. Brayshaw** (1991). "*An Architecture for Visualising the Execution of Parallel Logic Programming*". In Proceedings of The Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91). Sydney, Austria, 24-30 August, Vol. 2, pp. 870-876.

[BRA 91b] **M. Brayshaw et M. Eisenstadt** (1991). "*A Practical Graphical Tracer for Prolog*". International Journal of Man-Machine Studies, Vol. 35, N°5, pp. 597-631.

[BRE 84] **M. Bret** (juin 1984). "*L'image numérique animée*", Doctorat d'état, Université de Paris VIII.

[BRO 84] **M. H. Brown et R. Sedgewick** (1984). "*A System for Algorithm Animation*". In Proceedings of ACM SIGGRAPH '84, ACM Press, New York, pp. 177-186.

[BRO 85] **G. P. Brown, R. T. Carling, C. F. Herot, D. A. Kramlich et P. Souza** (1985). "*Program visualisation: graphic support for software development*". IEEE Computer, Vol. 18, N°8, pp. 27-35.

[BRO 88] **M. H. Brown** (1988). "*Exploring Algorithms Using Balsa II*". IEEE Computer, Vol. 21, N°5, pp. 14-36.

[BRO 91a] **M. H. Brown** (1991). " *Zeus: A System for Algorithm Animation and Multi-View Editing*". In Proceedings of IEEE Workshop on Visual Languages. Kobe, Japon, October, pp. 4-9.

[BRO 91b] **M. H. Brown et J. Hershberger** (1991). "*Color and Sound in Algorithm Animation*". Computer, Vol. 25, N°12, pp. 52-63.

[BRO 93] **M. H. Brown et M. A. Najork** (1993). "*Algorithm Animations Using 3D Interactive Graphics*". In Proceedings of UNIST'93 (Technical Report). DEC Systems Research Centre, Palo Alto, CA.

[BSD 88] **BSD Unix Distribution** (1988). "*vgrind*". C program, running on BSD Unix or derivatives, available with most UNIX system software. University of California at Berkeley.

[BUD 91] **T. Budd** (1991). "*An Introduction to Object-Oriented Programming*" Addison-Wesley.

[BUR 71] **N. Burtnyck et M. Wein** (1971). "*Computer : generated key-frame animation*". Journal of Society for Motion Picture and Television Engineers, 80, pp. 149-153.

[CAR 88] **Luca Cardelli** (1988). "*Building user interfaces by direct manipulation*". In proceeding of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada, pp. 152-166.

[CAT 72] **E. Catmull** (1972). "*A System for Computer Generated Movies*". In Proceeding ACM Annual Conference, New York, August.

[CEN 91] **CenterLine Software** (1991). "*ObjectCenter Reference*". Cambridge, MA: CenterLine Software, Inc.

[CHA 85] **R. Chandhok et al.** (1985). "*Programming Environments based on structure editing: The GNOME approach*". In Proceeding of the National Computer Conference (NCC' 85), AFIPS.

[CHA 90] **S. K. Chang** (1990). "*Principles of Visual Programming Systems*". Prentice Hall, New York.

[CHA 91] **R. Chandhok, D. Garlan, G. Meter, P. Miller et J. Pane** (1991). Pascal Genie. (version 1.0) San Diego, CA: Chariot Software Group. Pascal programming environment with integrated structure-driven editor, runs on Macintosh Computers. available from Chariot Software Group, San Diego, CA.

[CHE 76] **P. Chen** (1976). "*The entity Relationship Model - Toward a Unified View of Data*" . ACM Transaction on Database Systems, March.

[CIT 95] **W. Citrin, M. Doherty et B. Zorn** (1995). "*The Design of a Completely Visual Object-Oriented Programming Language (extended Abstract)*". In Visual Object-Oriented Programming : Concepts and Environments, Prentice-Hall, New York.

[COA 91] **P. Coad et E. Yourdon** (1991). "*Object-Oriented Analysis*". 2nd Edition, Yourdon Press Computing Series.

[CON 70] **K. Conrow et R. G. Smith** (1970). "*NEATER2: A PL/I Source Statement Reformatter*". Communications of the ACM, Vol. 13, N°11, pp. 669-675.

[COU 85] **P. Courtois** (1985). "*On Time and Space Decomposition of Complex Structures*". Communication of the ACM. Vol. 28, N°6, pp. 596-608.

[COU 90a] **J. F. Coudurier** (1990). "*La simulation des flux de production*". Rapport d'Etude N°105330, Centre Technique des Industries Mécaniques.

[COU 90b] **J. Coutaz** (1990). "*Interface Homme-Ordinateur, Conception et Réalisation*". Dunod Informatique, Bordas, Paris.

[COX 86] **B. J. Cox** (1986). "*Object-Oriented Programming, An Evolutionary Approach*". Addison Wesley.

[CUN 86] **W. Cunningham et K. Beck** (1986). "*A Diagram for Object-Oriented Programs*". Proceedings OOPSLA'86, Sept. 29 - Oct. 2, Portland, Oregon. SIGPLAN Notices, November, Vol. 21, N°11, pp. 361-367.

[CYP 93] **A. Cypher** (1993). "*Watch What I Do: Programming by Demonstration*". Cambridge, MA: Ed. MIT Press.

[DIG 92] **C. J. DiGiano et R. M. Baecker** (1992). "*Program Auralization: Sound Enhancements to the Programming Environment*". In Proceedings of Graphics Interface'92, Palo Alto, CA: Morgan Kaufmann, pp. 44-53.

[DUI 86a] **R. A. Duisberg** (1986). "*Animated graphical interfaces using temporal contraints*". In Proceedings of the ACM SIGGHI'86 Conference on Human Factors in Computing Systems, Boston, MA, April, pp. 131-136.

[DUI 86b] **R. A. Duisberg** (1986). "*Animated Graphical Interfaces Using Temporal Contraints*". PhD thesis, University of Washington, Seatle, WA, June. Technical Report 86-09-01.

[DUI 87] **R. A. Duisberg** (1987). "*Visual programming of program visualisations. A gestural interface for animating algorithms*". In IEEE Computer Society Workshop on Visual Languages, Linkoping, Sweden, August 1987, pp. 55-66.

[EDW 88] **A. D. N. Edwards** (1988). "*The design of auditory interfaces for visually disabled users*". ACM SIGCHI Conference on Human Factors in Computing Systems, pp. 83-88.

[EIS 88] **M. Eisenstadt et M. Brayshaw** (1988). "*The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming*". Journal of Logic Programming, Vol. 5, N°4, pp. 1-66.

[EIS 90] **M. Eisenstadt, J. Domingue, T. Rajan et E. Motta** (1990). "*Visual Knowledge Engineering*". IEEE Transactions on Software Engineering, Vol. 16, pp. 1164-1177.

[FEI 82] **S. Feiner, D. Salesin et T. Banchoff** (1982). "*Dial : A diagrammatic animation language*". IEEE Computer Graphics and Applications, Vol. 2, N°7, pp. 43-54.

[FEL 79] **S. I. Feldman** (1979). "*Make : A Program for Maintaining Computer Programs*". Software, Practice and Experience, Vol. 9, pp. 225-265.

[FLI 90] **S. Flinn et W. Cowan** (1990). "*Visualizing the Execution of Multi-processor Real-Time Programs*". In Proceedings of Graphics Interface, Halifax, Nova Scotia, pp. 293-300.

[FRA 92] **J. M. Francioni, L. Albright et J. A. Jackson** (1992). "*Debugging parallel programs using sound*". SIGPLAN Notices, Vol. 26, N°12, pp. 68-75.

[GAV 91] **W.Gaver, T. O'Shea et R. Smith** (1991). "*Effective Sounds in Complex Systems: The ARKola SImulation*". In Proceedings of Human Factors in Computing Systems (CHI `91). New Orleans, Louisiana, 27 April- 2 May, pp. 85-90.

[GES 75] **C. M. Geschke et J. G. Mitchell** (1975). "*On the Problem of Uniform References to data Structures*" SIGPLAN Notices, Vol. 10, N°6, pp. 31-42.

[GLI 84] **E. P. Glinert et S. Tanimoto** (1984). "*Pict: An Interactive Graphical Programming Language*". IEEE Computer, Nov., Vol. 7, N°25, pp. 7-25.

[GLI 90a] **E. P. Glinert** (1990). "*Visual Programming Environments : Paradigms and Systems*". IEEE Computer Society Press, Los Alamitos, CA.

[GLI 90b] **E. P. Glinert** (1990). "*Visual Programming Environments : Applications and Issues*". IEEE Computer Society Press, Los Alamitos, CA.

[GLI 92] **E. P. Glinert et D. N. Charles** (1992). "*Novis: A Visual Laboratory for Exploring the Design of Processor Arrays*". Journal of Visual Languages and Computing, Vol. 3, N°2, pp. 135-159.

[GOL 47] **H. H. Goldstein et J. Von Newmann** (1947) "*Planning and coding problems of an electronic computing instrument*". In : J. Van Newmann, Collected Works (A. H. Taub, ed.) McMillan, New York, pp 80-151.

[GOL 83] **A. Goldberg et D. Robson** (1983). "*Smalltalk-80, The language and its implementation*" Addison-Wesley, Publishing Company.

[GOL 84] **A. Goldberg** (1984). "*Smalltalk-80: The Interactive Programming Environments*". Addison-Wesley, Reading, MA.

[GOL 89] **D. R. Goldenson** (1989). "*The Impact of Structure Editing on Introductory Computer Science Education: The Results So Far*". ACM SIGCSE Bulletin, Vol. 21, N°3, pp. 26-29.

[GOL 90] **E. J. Golin et S. P. Reiss** (1990). "*The specification of Visual Language syntax*". Journal of Visual Languages and Computing, Vol. 1, N°2, pp. 141-157

[GOL 91] **E. J. Golin** (1991). "Tools review: Prograph 2.0 from TGS systems". Journal of Visual Languages and Computing, Vol. 2, N°2, pp. 189-194.

[GOU 84] **L. Gould et W. Finzer** (1984). "*Programming by rehearsal*". Byte, Vol. 9, N°6, pp. 187-210.

[GRA 85] **R. B. Grafton et T. Ichikawa** (1985) "*Special Issue on Visual Programming*". Eds. IEEE Computer, Vol. 18, N°6, pp 6-94.

[HAB 86] **A. N. Habermann et D. Notkin** (1986). "*Gandalf : Software Development Environment*". IEEE Trans. Software Engineering Vol. SE-12, N°12, pp. 1117-1127.

[HAI 59] **L. M. Haibt** (1959) "*A program to draw multi-level flow charts*". Proceedings of the Western Joint Computer Conference. San Francisco, California, 3-5 March, pp. 131-137.

[HAL 84] **D. C. Halbert** (1984). "*Programming by Example*". PhD thesis, University of California at Berkeley, BerKeley, CA.

[HEL 89] **E. Heltulla, A. Hyskykari et K.-J. Räihä** (1989). "*Graphical Specification of Algorithm Animations with ALLADIN*". In Proceedings of The Twenty-Second Annual Hawaii International Conference on System Science. New York: IEEE Computer Society, pp. 892-901.

[HEN 90a] **R. R. Henry, K. M. Whaley et B. Forstall** (1990a). "*The University of Washington Illustrating Compiler*". In Proceedings of The ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, New York, Vol. 25, N°6, pp. 223-233

[HEN 90b] **R. R. Henry, K. M. Whaley et B. Forstall** (1990b). "*The University of Washington Program Illustrator (UWPI)*". Seattle, WA: Computer Science and Engineering, University of Washington. Common Lisp/CLX source code for an automatic Pascal algorithm animation system, runs on Unix Workstations. available by anonymous ftp from june.cs.washington.edu as pub/uwpi.tar.Z.

[HER 96] **H. H. Hersey, S. T. Hackstadt, L. T. Hansen et A. D. Malony** (1996). "*Viz : A Visualization Programming System*". University of Oregon, Department of Computer and Information Science, Technical Report CIS. TR - 96. 05.

[HEW 77] **C. Hewitt et R. Atkinson** (1977). "*Parallelism and Synchronization in Actor System*". ACM Symposium on Principles of programming languages, 4 January, L.A.C.A.

[HIL 92] **D. D. Hils** (1992). "*Visual Language and computing Survey*". Journal of Visual Languages and Computing, Vol. 3, N°1, pp. 69-101.

[HIR 91] **M. Hirakawa, Y. Nishimura, M. Kado et T. Ichikawa** (1991). "*Interpretation of Icon Overlapping in Iconic Programming*". Proceedings of the 1991 IEEE Workshop on Visual Languages, Kobe, Japan, Oct. 8-11, pp. 40-60.

[HOR 93] **Radu Horaud, Olivier Monga** (1993). "*Vision par Ordinateur, Outils fondamentaux*". Traité de Nouvelles Technologies, série Informatique, Ed. Hermes.

[HUE 77] **J. Hueras et H. Ledgard** (1977). "*An Automatic Formatting Program for Pascal*". ACM SIGPLAN Notices, Vol. 12, N°7, pp. 82-84.

[HUR 78] **R. D. Hurrion et R. J .R. Secker** (1978). "*Visual interactive simulation, an aid to decision making*". Omega. UK. Vol. 6, N°5, pp 419-426.

[IGL 89] **I.G.L Technology** (1989). "*SADT un langage de Communication*". Eyrolles, Paris.

[ING 88] **D. Ingalls, S. Wallace, Y.Y. Chaw, F. Ludolph et K. Dayle** (1988). "*Fabrick - A visual Programming Environment*". Proc. OOPSLA'88, San Diego, pp. 176-190.

[KIM 86] **T. D. Kimura, J. W. Choi, et J. M. Mack** (1986). "*A Visual Language for Keyboardless Programming*". Technical Report WUCS-86-6, Department of Computer Science, Washington University, St. Louis, 1986.

[KNO 66a] **K. C. Knowlton** (1966). "*L^6: Bell Telephone Laboratories Low-Level Linked List Language*". Technical Information Libraries, Bell Laboratories, Inc., Murray Hill, New Jersey. 16 mm black and white sound film, 16 minutes.

[KNO 66b] **K. C. Knowlton** (1966). "*L^6: Part II. An Example of L^6 Programming*". Technical Information Libraries, Bell Laboratories, Inc., Murray Hill, New Jersey. 16 mm black and white sound film, 30 minutes.

[KNO 66c] **K. C. Knowlton** (1966). "*A Programmer's Description of L^6*". Communications of the ACM, Vol. 9, N°8, pp. 616-625.

[KNU 63] **D. E. Knuth** (1963). "*Computer-Drawn Flowcharts*". Communications of the ACM, Vol. 6, N°9, pp. 555-563.

[KNU 84] **D. E. Knuth**(1984). "*Literate Programming*". The Computer Journal, Vol. 27, N°2, pp. 97-111

[KRA 93] **E. Kraemer et J. T. Stasko** (1993). "The Visualization of Parallel Systems: an overview". Journal of Parallel and Distributed Computing, Vol. 18, pp. 105-117.

[LAN 93] **Ioan D. Landau** (1993) "*Identification et commande des systèmes*". Traité de Nouvelles Technologies, série Automatique, Ed. Hermes.

[LED 75] **H. F. Ledgard** (1975). "*Programming Proverbs*". Hayden, Rochell Park, New Jersey.

[LEW 87] **C. Lewis et G. M. Olson** (1987). "*Can principles of cognition lower the barriers to programming* ?" Empirical Studies of Programmers, Ablex, Vol. 2.

[LIE 87] **H. Lieberman** (1987). "*An Example Based Environment for Beginning Programmers*". In Artificial Intelligence and Education. Ablex, Norwood, N.J., pp. 135-151.

[LIE 89] **H. Lieberman** (1989). "*A Three-Dimensional Representation For Program Execution*". In Proceedings of The 1989 IEEE Workshop on Visual Languages. New York: IEEE Computer Society Press, pp. 111-116.

[LON 85] **R. L. London et R. A. Duisberg** (1985). "*Animating Programs using Smalltalk*". Computer, Vol. 18, N°8, pp. 61-71.

[MAR 77] **F. Martinez** (1977). "*Techniques de passage d'un dessein à un autre par déformation successives. Application à un système d'animation*", RR N°65, IMAG.

[MCC 87] **B. H. McCormick, T. A. DeFanti et M. D. Brown** (1987). "*Visualisation in scientific computing*". ACM Computer Graphics, Vol. 21, N°6.

[MCC 88] **J. McCormack et P. Asente** (1988). "*An Overview of the X Toolkit*", in Proceedings of the ACM SIGGRAPH Symposium on User Interface Software, Banff, Alberta, Canada. Oct., 17-19, ACM Press, pp 46-55.

[MCI 92] **D. W. McIntyre et E. P. Glinert** (1992). "*Visual tools for generating Iconic Programming Environements*". Proc. IEEE Workshop Visual Languages, Seattle, WA., Sept., pp. 162-168.

[MEL 88] **S. Mellor et S. Shlaer** (1988). "*Object-Oriented System analysis : Modelling the world in data*". Englewood Cliffs, N J : Yourdon Press.

[MEV 87] **A. Mevel et T. Guégen** (1987) "*Smallatlk-80*" Eyrolles Edition.

[MEY 88] **B. Meyer** (1988). "*Object-Oriented Software Construction*". Prentice Hall.

[MOH 88] **T. G. Moher** (1988). "*Provide: A process visualisation and debugging environment*". IEEE Transaction on Software Engineering, Vol. 14, N°6, pp. 849-857.

[MOR 85] **M. Moricono et D. F. Hare** (1985). "*Visualizing Program Designs Through Pegasys*". IEEE Computer, Vol. 18, N°8, pp. 72-85.

[MOS 93] **M. Mostefai** (1993). "*P.Os.T. : An architectural model for Interactive Simulation Interfaces Design*". IEEE/ SMC'93 Conference, Systems Engineering in the service of the Humans, le Touquet, France, Vol. 2, October 17-20, pp 550-555.

[MOS 94a] **M. Mostefai** (1994). "*Un Modèle d'Architecture oriente objet pour la conception de plates-formes de simulation interactive*". PhD thesis, Université des Sciences et Technologies de Lille, France.

[MOS 94b] **M. Mostefai et F. Van de Veire** (1994). "*An Object Oriented Architectural Model for Animating Algorithms*" Proceedings of the European Simulation Symposium, ESS'94, Istanbul, Turquie, Vol. 1, 9-10 Oct., pp. 107-111.

[MOU 90] **S. J. Mountford et W. W. Gaver** (1990). "*Talking and Listening to Computers*". In: The Art of Human-Computer Interface Design (Brenda Laurel, Eds.), Addison-Wesley, Reading, Massachusetts, pp. 319-334.

[MYE 83] **B. A. Myers** (1983). "*Incense: A System for Displaying Data Structures*". Computer Graphics, Vol. 17, N°3, pp. 115-125.

[MYE 87] **B. A. Myers** (1987). "*Creating dynamic interaction techniques by demonstration*". In proceeding of the 1987 Conference on Human Factors in Computing Systems and Graphic Interface, Toronto, Canada, April, pp. 271-278.

[MYE 88] **B. A. Myers, R. Chandhok et A. Sareen** (1988). "*Automatic Data Visualization for Novice Pascal Programmers*". In Proceedings of The IEEE Workshop on Visual Languages, Pittsburgh, PA, October, pp. 192-198. New York: IEEE Computer Society Press.

[MYE 90a] **B. A. Myers** (1990). "*Taxonomies of Visual Programming and Program Visualisation*". Journal of Visual Languages and computing, Vol. 1, pp. 97-123.

[MYE 90b] **B. A. Myers et all.** (1990). "*Garnet : Comprehensive Support for graphical, Highly-Interactive User Interfaces*". IEEE Computer, Vol. 23, N°11, pp. 71-85.

[MYE 93] **B. A. Myers, R. G. McDaniel et D. S. Kosbie** (1993). "*Marquise : Creating Complete User Interfaces by Demonstration*". Human factors in Computing Systems, Proceedings INTERCHI'93, Amsterdam, The Netherlands, April, pp. 293-300.

[NAS 73] **I. Nassi et B. Shneiderman** (1973). "*Flowchart Technique for Structured Programming*". ACM SIGPLAN Notices, Vol. 8, N°8, pp. 12-26.

[NEL 85] **G. Nelson** (1985). "*Juno, a constraint-based graphics system*". Computer graphics (SIGGRAPH'85), Vol. 19, N°3, pp. 235-243.

[NOR 81] **K. U. Nori, U. Ammann, K. Jensen, HH. Nägeli et C. Jacobi** (1981). "*Pascal P Implementation notes*". In Barraon, pp. 125 170.

[NOR 90] **C. D. Norton et E. P. Glinert** (October 1990). "*A Visual Environment for Designing and Simulating Execution of Processor Arrays*". IEEE Workshop on Visual Languages, Skokie, Illinois, pp. 227-232.

[NSI 85] **NSIA (National Security Industry Association)** (1985). Proceedings First Joint DoD-Industry Symp. On the STARS Program, San Diego (Calif.), 30 Avril au 2 Mai.

[PAP 95] **A. Papantonakis et J. H. K. Peter** (March 1995). "*Syntax and Semantics of Gql, a Graphical Query Language*". Journal of Visual Languages and Computing, Vol. 6 ,N°1, pp. 3-25.

[PAR 72] **D. L. Parnas** (1972). "*On the Criteria to be Used in Decomposing Systems into Modules*" Communication of the ACM, Vol. 5, N°12, pp. 1053-1058.

[PRI 90] **B. A. Price** (1990). "A *Framework for the Automatic Animation of Concurrent Programs*". M.Sc. Thesis, Dept. of Computer Science, University of Toronto, Canada M5S 1A1.

[PRI 91] **B. A. Price et R. M. Baecker,** (1991). "*The Automatic Animation of Concurrent Programs*". In Proceedings of The First International Workshop on Computer-Human Interfaces,. Moscow, USSR: ICSTI, pp. 128-137.

[PRI 93] **B. A. Price, R. M. Baecker et I. S. Small** (1993). "*A Principled Taxonomy of software Visualisation*". Journal of Visual Languages and Computing, Vol. 4, pp. 211-266.

[REE 81] **W. T. Reeves** (1981). "*In betweening for Computer Animation Utilizing Moving Point Constraints*". Computer graphics, Vol. 15, N°3, pp. 262-269.

[REI 85] **S. P. Reiss** (1985). "*Pecan: Program Development Systems that support Multiple Views*". IEEE Transaction on Software Engineering, Vol. SE-11, N°3, pp. 276-285.

[REI 90] **S. P. Reiss** (1990). "*Interacting with the FIELD Environment*". Software Practice and Experience, Vol. 20, (S-1): 89.

[REY 82] **C. W. Reynolds** (1982). "*Computer animation with Scripts and Actors*". Computer graphics, Vol. 16, N°3, pp. 269-296.

[ROC 89] **A. Rochfeld, J. Morejon** (1989). "*La Méthode Merise*". Gamme Opératoire, Paris, les Editions d'Organisation.

[ROM 89] **G.-C. Roman et K. C. Cox** (1989). "*A Declarative Approach to Visualizing Concurrent Computations*". Computer, Vol. 22, N°10, pp. 25-36.

[ROM 90] **G.-C. Roman et H. C. Cunningham** (1990). "*Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency*". IEEE Transactions on Software Engineering, Vol. 16, pp. 1361-1373.

[ROM 92a] **G.-C. Roman, K. C. Cox, J. Y. Plun et C. D. Wilcox,** (1992). "*From Proofs to Pictures*". 14 minute colour S-VHS videotape with sound. St. Louis, MO 63130-4899: Dept. of Computer Science, Washington University.

[ROM 92b] **G.-C Roman, K. C. Cox, C. D. Wilcox et J. Y. Plun** (1992). "*Pavane: a System for Declarative Visualization of Concurrent Computations*". Journal of Visual Languages and Computing, Vol. 3, N°2, pp. 161-193.

[ROM 93] **G. C. Roman et K. C. Cox** (Décembre 1993) "*A Taxonomy of Program Visualization Systems*". IEEE Computer, pp. 11-24.

[ROY 76] **P. Roy et R. St. Denis** (1976). "*Linear Flowchart Generator for a Structured Language*". ACM SIGPLAN Notices, Vol. 11, N°11, pp. 58-64.

[RUB 89] **R. V. Rubin** (1989). "*A Logical Basis for Programming by Demonstration*". PhD thesis, Brown University, Providence, RI, May.

[SCA 89] **D. A. Scanlan** (1989). "*Structured Flowcharts Outperform Pseudocode: An Experimental Comparison*". IEEE Software, Vol. 6, N°7, pp. 28-36.

[SCH 91] **A. Schürr** (1991). "*PROGRES: A VHL-Language Based on Graph Grammars*". Proc. 4th Int. Workshop on Graph-Grammars and Their Application to Computer Science, LNCS 532, Springer Verlag, pp. 641-659, also: Technical Report AIB 90-16, RWTH Aachen, Germany, 1990.

[SED 83] **R. Sedgewick** (1983). "*Algorithms Book*". Addison-Wheley.

[SEN 90a] **B. Senach** (mars 1990) "*Evaluation ergonomique des interfaces homme-machine : une revue de la littérature*". Rapport de recherche, INRIA, Sophia Antipolis, n°1180.

[SEN 90b] **B. Senach** (1990) "*Evaluation ergonomique des interfaces homme-machine : du prototypage aux systèmes experts*". Actes du congrès Ergo-IA'90 : Ergonomie et Informatique avancée, Biarritz, France.

[SHN 77] **B. Schneiderman et al.** (1977). "*Experimental investigations of the Utility of Detailed Flowcharts in Programming*". Comm. ACM, June, pp. 373-381

[SHO 85] **K. Shoemake** (1985). "*Animating Rotation with Quaternion Curves*". SIGGRAPH'85, pp. 245-254.

[SHU 88] **N. C. Shu** (1988). "*Visual Programming*". New York: Van Nostrand Reinhold.

[SMI 86] **R. B. Smith** (1986). "*The Alternative Reality Kit: An animated environment for creating interactive simulations*". In IEEE Computer Society Workshop on visual Languages, Dallas, TX, June, pp. 99-106.

[SMI 87] **R. B. Smith** (1987). "Experiences with the Alternative Reality Kit, an example of the tension between literalism and magic". In Proceeding of ACM SIGGCHI'87 Conference on Human Factors in Computing Systems, Toronto, Canada, April, pp. 61-67.

[SMI 91] **S. Smith, R. D. Begeron et G. G. Grinstein** (1991). "*Stereophonic and surface sound generation for explanatory data analysis*". ACM SIGCHI'91 Conference on Human Factors in Computing Systems, pp. 125-132.

[STA 80] **T. A. Standish** (1980) "*Data Structure Techniques*". Addison-Wheley.

[STA 89] **J. T. Stasko** (1989). "*TANGO: A Framework and System for Algorithm Animation*" PhD Dissertation. Brown University, Department of Computer Science, Providence , RI 02912, N° CS-89-30, May 1989, 257 pages.

[STA 90] **J. T. Stasko** (1990). "*Tango: A Framework and System for Algorithm Animation*". IEEE Computer, Vol. 23, N°9, pp. 27-39.

[STA 91] **J. T. Stasko** (1991). "*Using Direct Manipulation to Build Algorithm Animations by Demonstration*". In Proceedings of the ACM SIGCHI'91 Conference on Human Factors in Computing Systems , New Orleans, May 1991, pp. 307-314.

[STA 92] **J. T. Stasko** (1992). "*Polka animation designer's package*". Technical report, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280.

[STA 93] **J. T. Stasko et J. F. Wehrli** (1993). "*Three-Dimensional Computation Visualization*". In Proceedings of IEEE/CS Symposium of Visual Languages, Bergen, Norway, 24-27 August.

[STR 88] **P. S. Strauss** (1988). "*BAGS : the Brown Animation Generation System*". Ph. D. Thesis, Technical Report CS-88-22, Computer Science Department, Brown University, Providence, RI, May.

[SUT 63] **I. E. Sutherland** (1963). "*SketchPad : A man-machine graphical communication system*". AFIPS Spring Joint Computer Conference, N°23, pp 329-346.

[SWI 86] **D. C. Swinehart et al.** (1986). "*A structural View of the Cedar Programming Environment*" ACM Trans. Programming Languages and Systems, Vol. 8, N°4, pp. 419-490.

[SYM 85] **Symbolics** (1985). Inc. , "*S-Dynamics*". Inc., Cambridge, MA.

[SZM 97] **P. Szmal et J. Fancik** (1997). "*Algorithm Animation and Debugging with the WinSanal System*". 15th IASTED International Conference, Applied Informatics, Innsbruck, Feb. 18-20.

[TAN 90] **S. L. Tanimoto** (1990). "*Towards a theory of Progressive Operators for Live Visual Programming Environments*". Proceedings of the 1990 IEEE Computer Society Workshop on Visual Languages, Skokie, Illinois, Oct. 4-6, pp. 80-85.

[TEI 81] **T. Teitelbaum et T. Reps** (1981). "*The Cornell Program Synthesizer : A syntax-directed Programming Environment*". Communication ACM, Vol. 24, N°9, pp. 563-573.

[THA 85] **D. Thalmann et N. Magnenat** (1985). "*Controlling Evolution and Moving Using the CINEMIRA-2 Animation Sublanguage*". In Computer generated images. Springer, pp. 249-259.

[TUF 83] **E. R. Tufte** (1983). "*The Visual Display of quantitative Information*". Graphics Press, Cheshire, CT.

[TUF 90] **E. R. Tufte** (1990). "*Envisionning Information*". Graphics Press, Cheshire, CT.

[VAN 95] **F. Van de Veire** (1995). "*An Interactive and Convivial Object Oriented Simulation for Animation Algorithms*". Modelling and Simulation, ESM'95 (European Simulation Multiconference), Prague, République Tchèque, 5-7 Juin, pp. 599-603.

[VAN 96a] **F. Van de Veire** (1996). "*Modularly Animated Algorithms*". CESA'96, IMACS Multiconference IEEE/ SMC Symposium on Modelling, Analysis and Simulation, Lille, France, Vol. 2, July 9-12, pp 670-675.

[VAN 96b] **F. Van de Veire** (1996). "*Program Visualization and Visual Programming*" Modelling and Simulation, ESM'96 (European Simulation Multiconference), Budapest, Hungry, 2-5 Juin.

[VOS 86] **G. M. Vose et G. Williams** (1986). "*LabVIEW : Laboratory Virtual Instrument Engenieering Workbench*". Byte, N°11, pp. 84-96.

[WIL 91] **C. A. Williams** (1991). "*Retargetable Visual Programming Langage*". Ph.D. thesis, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM.

[WIR 71] **N. Wirth** (1971). "*The Design of a Pascal Compiler*". Software - Practice and Experience, Vol. 1, N°4, pp. 309-333.

[YAN 94] **S. Yang et M. Burnett** (1994), "*From Concrete Forms to Generalized Abstractions through Perspective-Oriented Analysis of Logical Relationships*". IEEE Symposium on Visual Languages, St. Louis, MO, October 4-7, pp. 6-14.

[YOU 79] **E. N. Yourdon et L. L. Constantine** (1979). "Structured *Design : Fundamentals of a Discipline of Computer Program and System Design*". Prentice-Hall, Englewood Cliffs (N. J.).

[YOU 89] **E. N. Yourdon** (1989). "Modern Structured Analysis". Yourdon Press, Computing Series.

[YOU 95a] **M. Youg, D. Argiro et S. Kubica** (1995). "*Cantata : Visual Programming Environment for the Khoros System*". Computer graphics, Vol. 29, N°2, May, pp. 22-24.

[YOU 95b] **M. Youg, D. Argiro et J. Worley** (1995). "*An Object Oriented Visual Programming Language Toolkit*". Computer graphics, Vol. 29, N°2, May, pp. 25-28.

[ZIM 88] **M. Zimmermann, F. Perrenoud et A. Schiper** (1988). "*Understanding Concurrent Programming through Program Animation*". In H. L. Dershen (Ed.), Proceedings of The Nineteenth ACM SIGCSE Technical Symposium on Computer Science Education, pp. 27-35.