Institut National Polytechnique de Lorraine

Centre de Recherche en Informatique de Nancy

THESE

Présentée pour l'obtention du grade de

DOCTEUR-INGENIEUR EN INFORMATIQUE

par

Jean-Pierre JACQUOT

ETUDE D'UN OUTIL D'ASSISTANCE A LA CONCEPTION METHODIQUE DE PROGRAMMES

Réalisation et évaluation d'une maquette

Soutenue publiquement le 15 Septembre 1984

devant la commission d'examen:

MM. FINANCE J.P. Président

DERNIAME J.C.

GUYARD J.

KAHN G.

Institut National Polytechnique de Lorraine

Centre de Recherche en Informatique de Nancy

THESE

Présentée pour l'obtention du grade de

DOCTEUR-INGENIEUR EN INFORMATIQUE

par

Jean-Pierre JACQUOT

ETUDE D'UN OUTIL D'ASSISTANCE A LA CONCEPTION METHODIQUE DE PROGRAMMES

Réalisation et évaluation d'une maquette

Soutenue publiquement le 15 Septembre 1984

devant la commission d'examen:

MM.	FINANCE J.P.	Président
	DERNIAME J.C.	
	GUYARD J.	
	KAHN G.	
	KING P.	
	MOHR R.	

REMERCIEMENTS

Je remercie vivement Jean-Pierre FINANCE pour l'honneur qu'il me fait en présidant cette thèse. Par sa gentillesse, sa disponibilité, l'intérêt constant qu'il a porté à mon travail et sa bienveillante exigence, il a su donner à mon ouvrage la rigueur nécessaire.

Je dois énormément à Jacques GUYARD. Le système présenté dans cette thèse est le fruit d'un véritable travail d'équipe entre nous deux. Pour son amitié, ses compétences, sa disponibilité, son expérience et la confiance dont il a toujours tenu à me faire profiter: Merci.

Je remercie Gilles KAHN, un des pionniers de la recherche sur les environnements de programmation, de m'avoir consacré de son précieux temps et d'avoir accepté de venir de loin pour participer au jury.

Lors de son séjour à Nancy, Peter KING a apporté un point de vue nouveau sur les travaux de notre équipe. Qu'il soit ici remercié pour sa participation au jury.

Bien que son domaine de recherche soit différent du mien, Roger MOHR à su s'intéresser à mon travail. Je le remercie d'avoir accepté de juger cette thèse.

Je remercie Jean-Claude DERNIAME pour l'intérêt qu'il porte à ce travail et pour sa participation au jury.

Mes remerciements vont aussi à ceux dont l'efficacité à permis de résoudre bien des problèmes techniques lors de la réalisation, et surtout à Bertrand MELESE, de l'INRIA, qui fut notre contact amical et toujours disponible dans notre utilisation de MENTOR.

Je remercie Maryse QUERE et Jean-Michel HOC pour l'aide irremplaçable qu'il nous ont apportée lors de la mise au point d'EDME: ils ont en effet eu le redoutable privilège d'en être les premiers utilisateurs.

Le texte a été amélioré grâce à la lecture attentive d'Alain QUERE, les photographies ont été réalisées par François SCHWAAB; je les remercie vivement.

Le projet Maiday a pu être réalisé grâce au soutien du Greco de Programmation du C.N.R.S. qui nous a fourni les moyens matériels et financiers pour accéder aux outils logiciels utilisés, et au soutien de l'A.D.I. (contrat n[81/534).

Enfin, que Martine trouve ici toute ma gratitude pour l'important et ingrat travail de correction qu'elle a fourni et surtout, l'indéfectible soutien qu'elle m'a apporté tout au long de ce travail.

A tous: Merci.

Table des Matières

Chapitre 1

Introduction

1.1 Présentation

Cette thèse est la relation d'une expérience menée depuis deux ans à Nancy dans le domaine de la programmation. L'objet de cette expérience est simple: construire la maquette d'un environnement de programmation, **Maiday**. Son originalité réside essentiellement dans la focalisation du travail sur le processus de construction des programmes.

Les environnements actuellement construits mettent l'accent sur la production et la gestion de gros logiciels en amplifiant, grâce à des outils de plus en plus complexes, la puissance de travail du programmeur. Ce programmeur est un professionnel, rompu à l'utilisation de l'environnement. Les outils sont alors développés pour faciliter la mise en oeuvre de techniques sophistiquées (édition syntaxique, mise au point symbolique, mesures de complexité, de performance, ...) et pour permettre la gestion de structures complexes (les versions, les jeux de tests, les projets, les équipes de programmeurs, ...). Notre point de départ est différent: beaucoup de programmeurs sont en fait des novices (étudiants, programmeurs occasionnels, ...) et l'outil doit moins amplifier une puissance que *guider, aider, conseiller*, prendre à sa charge les tâches annexes et être d'un abord très facile.

Deux axes d'étude se dégagent alors:

1. définir les fonctionnalités d'un tel environnement (domaine de la méthodologie de la programmation).
2. définir et mettre au point un interface de qualité entre l'outil et l'utilisateur (domaine de l'ergonomie).

Enfin, un troisième axe concerne les techniques utiles pour développer, mettre au point et valider une maquette d'environnement de programmation (domaine du génie logiciel).

La définition des fonctionnalités d'un système est toujours une étape délicate qui nécessite un support solide pour être menée à bien. Le guide que nous avons utilisé est issu de l'analyse suivante:

> Les difficultés de la programmation sont essentiellement liées à la multiplicité des entités manipulées et de leurs relations mutuelles. Pour maîtriser cette complexité, il existe des moyens: sur le plan de la conception, les méthodes de programmation permettent d'expliciter les relations entre entités (les objets, les modules, ...), de découvrir celles qui sont nécessaires ou superflues et de guider l'analyse (décomposition en sous-problèmes, modularisation, ordre dans les définitions, ...). Sur le plan de

l'expression des solutions, les constructions syntaxiques (modules, instructions structurées, ...) et les contrôles statiques (syntaxe, typage, ordonnancement, ...) facilitent le travail. Mais une conséquence de ces techniques est qu'elles apparaissent de plus en plus comme des contraintes pénibles pour le programmeur. Afin de supprimer cet effet déplaisant, nous pensons que la méthode, la sémantique, la syntaxe et les contrôles qui leur sont associés devraient être directement intégrés à l'outil de façon à se transformer en une aide effective pour l'utilisateur.

Ainsi, les fonctionnalités sont inspirées par l'analyse de la méthode de programmation, de ses contraintes et de ses impératifs.

Dans un système hautement interactif, l'utilisateur devient une *composante* importante qu'il faut prendre en compte dès la conception. L'utilisateur auquel nous destinons Maiday est un programmeur novice, aussi avons-nous centré notre étude sur la simplicité d'emploi que doit présenter l'environnement. La simplicité intervient sur trois plans:

Le poste de travail. Celui de Maiday est relativement classique (écran alphanumérique, clavier, imprimante) et nous avons utilisé les techniques classiques (multi-fenêtrage, éditeur video, formateur).

Les aides à l'utilisation. Au delà des habituelles aides en-ligne, nous avons surtout développé l'intégration des différentes fonctionnalités au sein des commandes mises à la disposition du programmeur. Nous pensons en effet que l'aide principale qu'apporte Maiday est la prise en charge automatique des aspects non créatifs de la construction de programmes. Ceci va de l'activation des différents contrôles au guidage de l'utilisateur dans le processus de construction.

La cohérence des concepts. Souvent, le programmeur est obligé de manipuler simultanément plusieurs niveaux de concepts: ceux du problème à résoudre, ceux de la méthode employée, ceux du langage formel et ceux de l'outil utilisé. Nous essayons avec Maiday de *fusionner* ces niveaux, au moins les trois derniers. Pour ce faire, les niveaux sont présentés selon la même structure qui correspond à la vue la plus "logique" possible. Ainsi, les commandes ne s'adressent qu'à cette vue; l'outil a seul la charge de maintenir les liens, en particulier avec la structure syntaxique du texte.

Nous souhaitons que Maiday soit vu comme une *maquette*, c'est à dire comme un système permettant d'expérimenter rapidement les idées, autorisant de nombreuses possibilités de réglages (en allant de la disposition des fenêtres sur l'écran au degré de rigidité des contraintes et la directivité des dialogues), et aussi favorisant l'extrapolation vers des systèmes opérationnels. En plus des trois critères techniques précédents, nous nous imposons de mettre rapidement Maiday entre les mains d'utilisateurs n'appartenant pas à l'équipe de développement. Cette exigence est essentielle pour obtenir des critiques efficaces sur la maquette. Afin de satisfaire à ces quatre contraintes, nous avons respecté quelques principes:

- toujours aller au plus simple et au plus naturel
- développer des solutions suffisamment générales
- utiliser les fonctionnalités des outils (logiciels et matériels)sophistiqués.

Le dernier principe nous a amené à considérer le développement de Maiday comme une couche supplémentaire à un outil, MENTOR en l'occurrence. Nous pensons que la réalisation de logiciels d'application sera de moins en moins tributaire des langages de programmation mais plutôt d'outils paramétrables (de *méta-outils*): une application sera composée d'un *noyau* généré par un méta-outil et *d'extensions* l'enveloppant. Le développement de Maiday peut être considéré comme une mise à l'épreuve en vraie grandeur de cette hypothèse.

Nous avons adopté pour réaliser Maiday une démarche expérimentale. Trois étapes s'y distinguent:

1. Nous nous dotons d'un cadre théorique, issu:
   - des recherches Nancéennes sur la programmation pour ce qui concerne la méthode de programmation
   - d'une analyse sur l'interaction entre l'utilisateur et les outils que nous connaissons. Nous nous efforçons alors de mettre en évidence les hypothèses qui sous-tendent la conception de Maiday.
2. L'étape suivante consiste en la définition, la réalisation et la validation des fonctionnalités de Maiday. Le point crucial est ici la validation, tant technique (correction et qualité du logiciel) que logique (validité des options retenues), de l'implantation vis à vis des hypothèses précédentes. Cette étape est scindée en deux parties: la validation technique qui est réalisée par le transport du système sur une autre machine, ce qui permet de contrôler les techniques de développement employées; quant à la validation logique, elle est réalisée par l'observation de programmeurs utilisant Maiday. Cette seconde partie bénéficie du fait que Maiday sert de *plate-forme expérimentale* à des psychologues qui étudient les stratégies mises en oeuvre par les programmeurs.
3. La dernière étape consiste en une exploitation critique des observations effectuées lors de la validation. Le champ d'étude de Maiday (techniques de génie logiciel, relations homme-machine, méthodes de programmation) se prête mal à une quantification des résultats. Néanmoins, une critique qualitative permet d'estimer la validité des hypothèses et de les affiner substantiellement.

Cette démarche est naturellement cyclique. Un premier cycle a déjà permis d'améliorer la version initiale de Maiday (grossièrement, un sous-ensemble de celle qui est présentée ici).

1.2 Film d'une session

Nous présentons ici quelques photographies commentées d'une session avec Maiday. Nous avons délibérément choisi de ne présenter maintenant ni la méthode implantée ni le langage, MEDEE, qui sont utilisés pour construire des algorithmes avec Maiday (nous renvoyons pour cela le lecteur au chapitre 3,

sections 1.3 et 1.4). En effet, la raison d'être d'une maquette est de visualiser le plus rapidement possible ce que sera le système final!

L'énoncé du problème résolu par l'algorithme est le suivant:

> Transformer un flot d'entrée composé de chaînes de caractères de longueur variable en un flot de sortie composé de paquets de caractères de longueur fixe. Chaque paquet est formé par la concaténation des chaînes d'entrée. Le programme s'arrête lorsqu'une chaîne spéciale ("fin de transmission") apparaît.

Le commentaire situé en regard de chaque photographie se compose de trois parts: une phrase situant la vue, la liste des dernières entrées de l'utilisateur (encadrée par <>) avec un commentaire et enfin une explication plus détaillée sur la vue.

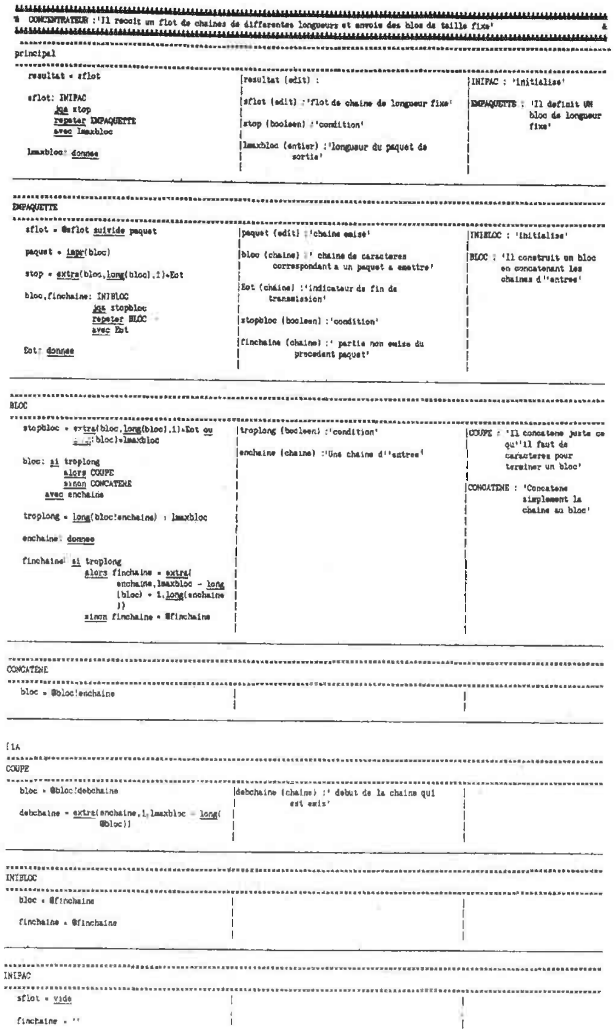


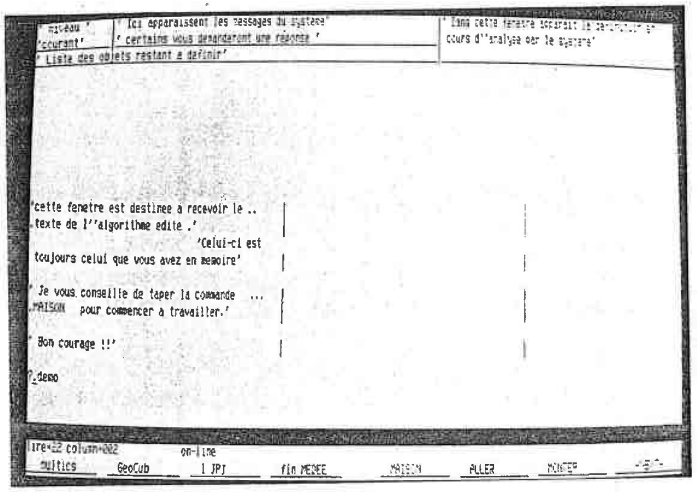
Figure 1. Algorithme MEDEE

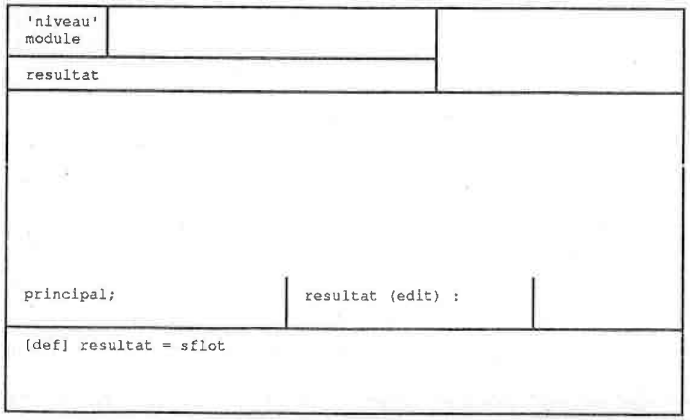
```
<><><><><><>
<> .DEMO  <>
<><><><><><>
```

Nous demandons la signification de chaque fenêtre. La fenêtre inférieure située au bas de l'écran est réservée aux entrées de l'utilisateur. Le curseur s'y positionne automatiquement.

```
<><><><><><><><><><><>
<> .CRALG          <>  { CRéer un ALGorithme }
<> CONCENTRATEUR   <>  { Nom de l'algorithme   }
<> 'Il recoit ...' <>  { Description informelle}
<> .CRDEF          <>  { CRéer une DEFinition  }
<><><><><><><><><><><>
```

Nous débutons la création d'un nouvel algorithme. EDME nous demande de le nommer et d'en donner une description informelle. Ceci fait, il initialise la construction en créant un module (appelé **PRINCIPAL**) qui comporte un objet prédéfini, **résultat**, à définir. EDME passe directement sur la couche Module. Nous définissons alors **résultat**.





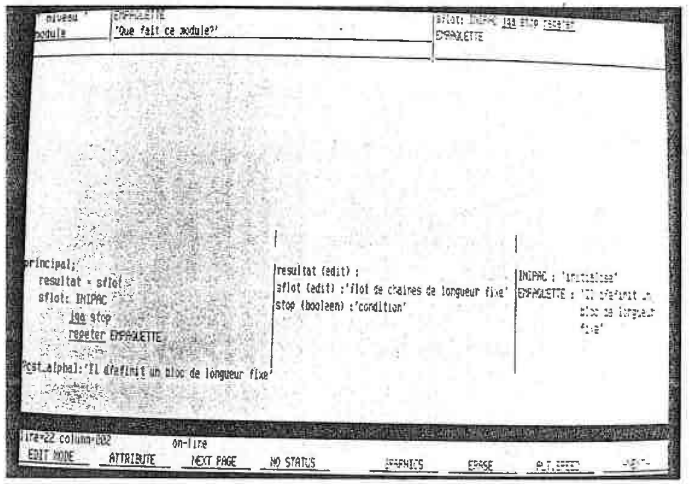
```
<><><><><><><><><><><><><><>
<> .CRDEF                  <>
<> sflot: INIPAC jqa stop <>
<>   repeter EMPAQUETTE    <>
<> 'il définit un ...'     <> { Description informelle de EMPAQUETTE }
<><><><><><><><><><><><><><>
```

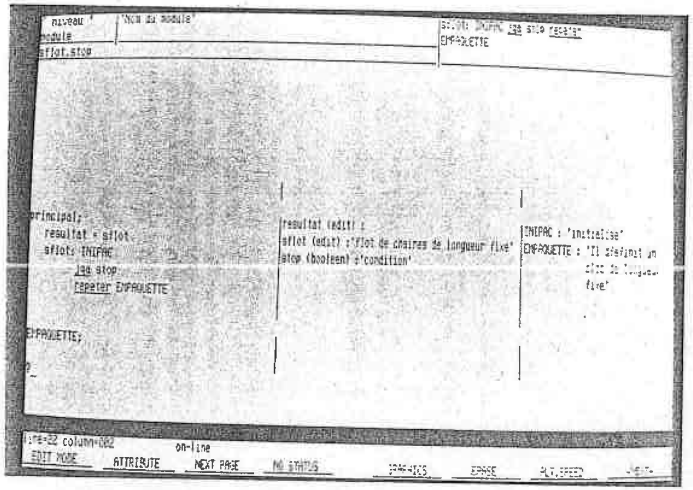
Lors de l'étape précédente **sflot** a été introduit. EDME nous a demandé son type et un commentaire pour construire sa définition informelle. Les définitions étant correctes, celles-ci ont été intégrées au module. Notez que la liste des objets à définir a été actualisée. Nous définissons alors **sflot**. La définition formelle apparaît en haut à droite; ceci indique qu'elle est syntaxiquement correcte et que les contrôles contextuels et méthodologiques sont en cours. Le module **EMPAQUETTE** ayant été introduit, EDME demande un commentaire.

note: la photographie est prise au cours de la remise en état de l'écran. Les définitions sont insérées dans le module, les messages vont être effacés.

```
<><><><><><><><><>
<> .ALLER      <> { Positionnement sur un module }
<> EMPAQUETTE <>
<><><><><><><><><>
```

Le module **PRINCIPAL** a été complété : i.e. tous les objets introduits ont été définis formellement. Nous poursuivons alors en construisant le module **EMPAQUETTE**. Notez comment le texte du module précédent a été formaté pour accroître la lisibilité. Notez aussi qu'EDME a calculé la liste des objets à définir dans le module **EMPAQUETTE**.



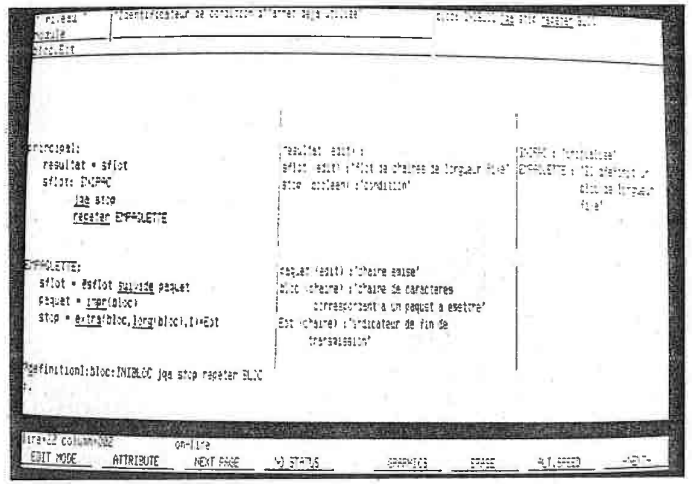


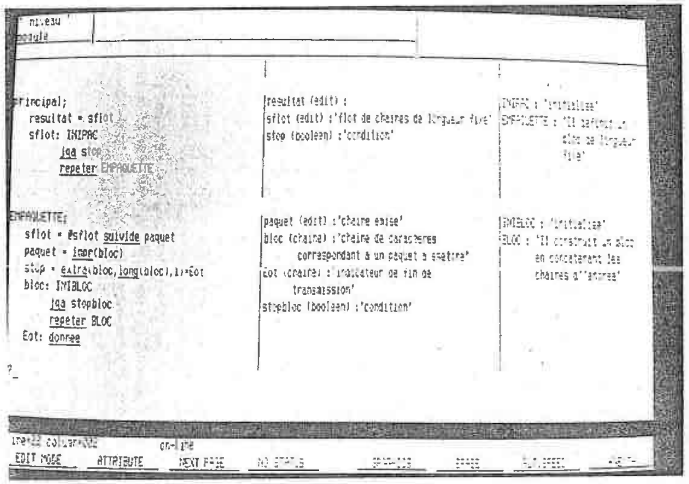
```
<><><><><><><><><><><><><><><><><><><>
<> .CRDEF                          <>
<> bloc : INIBLOC jqa stop         <>
<>  repeter BLOC                   <>
<><><><><><><><><><><><><><><><><><><>
```

Nous avons construit une bonne partie de **EMPAQUETTE** depuis la vue précédente ! Le symbole **@** devant **sflot** dénote *l'ancienne valeur*, i.e. celle qui a été calculée à l'itération précédente. Cette différentiation des valeurs est fondamentale en MEDEE qui est un langage à *affectation unique*. La définition itérative d'un objet est en fait la construction d'une **suite de valeurs**. Une contrainte de MEDEE en découle : tout identificateur doit être unique. EDME nous signale ici que cette contrainte est violée. La définition est donc refusée et, bien évidemment, ne se trouve pas intégrée au module.

```
<><><><><><><><><><><><>
<> .CRDEF              <>
<> Eot : donnee        <>
<><><><><><><><><><><><>
```

Le module **EMPAQUETTE** est complet. Ceci est indiqué par l'absence d'objets à définir dans la fenêtre. Les définitions informelles de **stopbloc** et **INIBLOC** sont identiques à celles de **stop** et de **INIPAC**. Ceci provient de ce qu'elles sont automatiquement générées par EDME qui les infère du contexte d'utilisation des identificateurs.



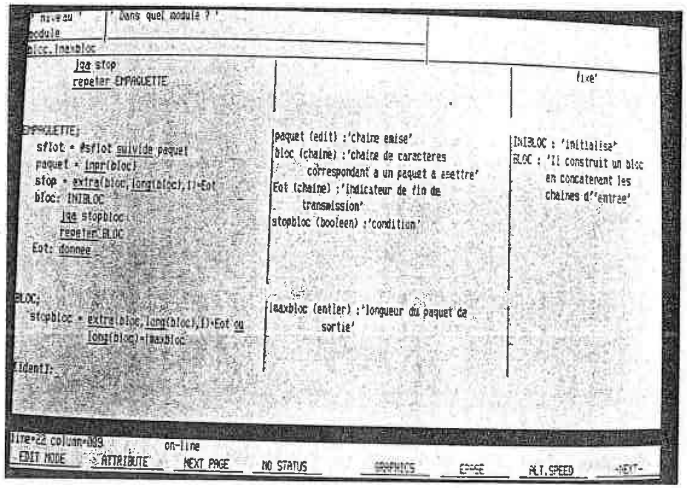


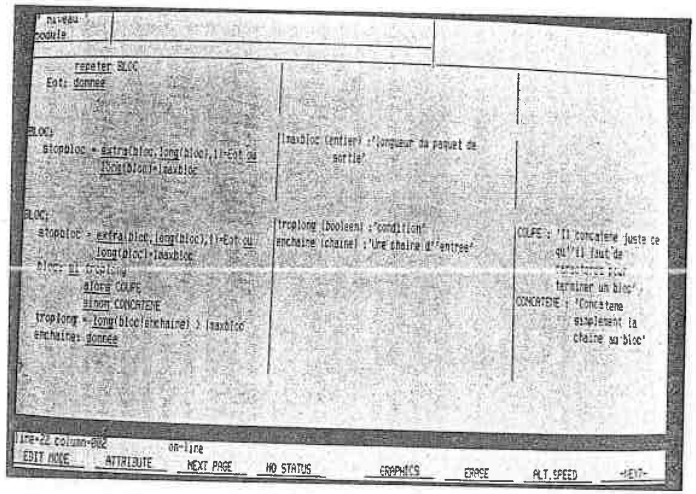
```
<><><><><><><><><><>
<> .REMONTER     <> { Déplacement d'une définition }
<> lmaxbloc      <>
<><><><><><><><><><>
```

La démarche déductive nous impose d'introduire **lmaxbloc** dans un module itéré alors que sa valeur y reste constante. EDME offre une commande de déplacement de définition (actuellement limitée aux définitions informelles) d'un module vers un autre.

```
<><><><><><><><><><><><><><>
<> .CRDEF                  <>
<> enchaine : donnee       <>
<><><><><><><><><><><><><><>
```

Nous avons complété **BLOC** (temporairement). Notez quer le module **BLOC** apparaît deux fois sur l'écran. Ceci est dû au fait que le module affiché doit toujours être l'image exacte du module mémorisé. Comme la commande précédente (.REMONTER) a modifié **BLOC**, EDME a provoqué un réaffichage complet du *nouveau* module **BLOC**.



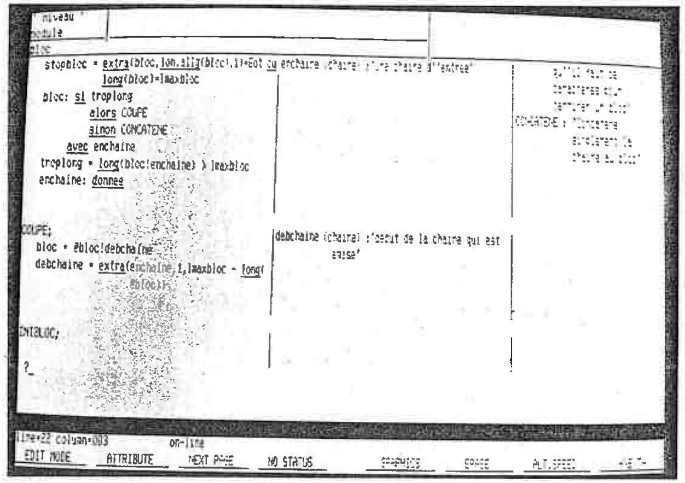


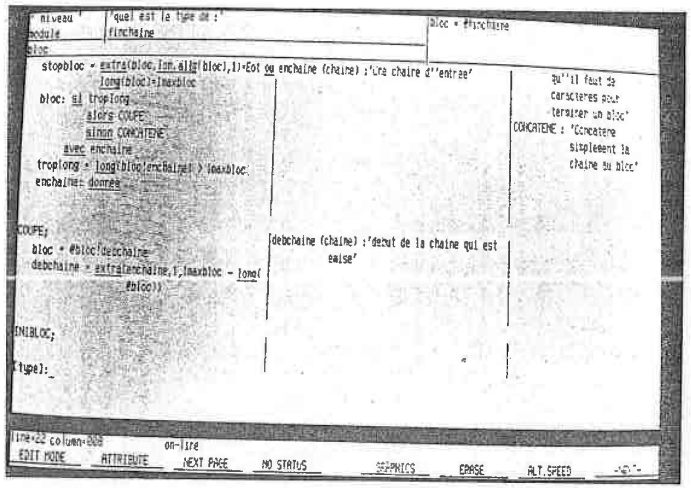
```
<><><><><><><>
<> .ALLER    <>
<> INIBLOC   <>
<><><><><><><>
```

Nous ouvrons un module d'initialisation. Notez que la liste des objets à définir ne comporte que les objets récurrents de **BLOC**. EDME calcule cette liste à partir d'une analyse des récurrences, nous déchargeant ainsi de ce travail souvent délicat.

```
<><><><><><><><><><><><><><><>
<> .CRDEF                    <>
<> bloc = @finchaine         <>
<><><><><><><><><><><><><><><>
```

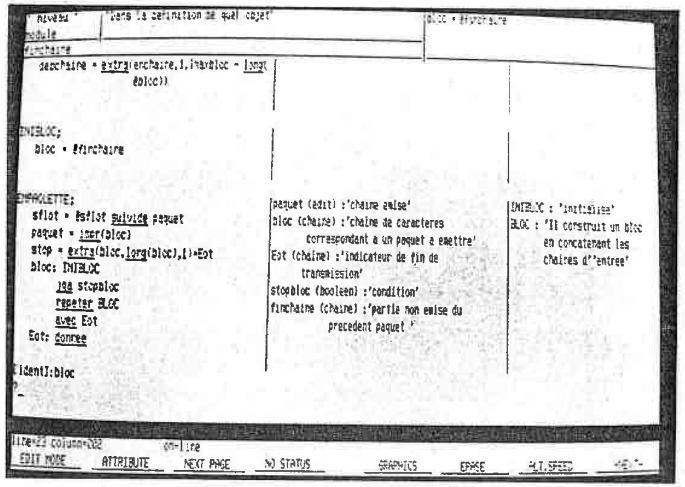
Nous introduisons un nouvel objet : **finchaine**. Comme il est précédé de **@**, il est récurrent. Or **INIBLOC** n'est pas un module itératif, donc **finchaine** ne peut y être défini. Pour résoudre ce problème, EDME calcule dans la hiérarchie des modules le plus proche module itératif englobant pour y placer la définition de **finchaine**. Dans notre algorithme, c'est **EMPAQUETTE** qui recevra la définition de **finchaine**. Ce module, que nous avions précédemment complété, redevient donc incomplet.





```
<><><><><><><><>
<> .ALLER      <>
<> EMPAQUETTE <>
<> .INSERER    <>
<> finchaine   <> { Objet à insérer }
<> bloc        <> { Définition où insérer }
<><><><><><><><>
```

Nous réouvrons **EMPAQUETTE** pour le compléter. La définition de **finchaine** utilise une facilité syntaxique de MEDEE : plusieurs objets peuvent être définis par la même définition itérative lorsque les domaines d'itérations sont identiques. Cette insertion rend donc **BLOC** de nouveau incomplet. **INIBLOC** reste complet pour l'instant car EDME ne peut encore déterminer que **finchaine** est récurrent dans **BLOC**.



1.3 Plan de lecture

Cette thèse est constituée de trois volets traitant de la conception de **Maiday**, de sa réalisation et d'une discussion critique des résultats obtenus.

1.3.1 Premier volet : la Conception

Cette partie présente un "cahier des charges" informel de Maiday. Nous y précisons le champ d'étude abordé ainsi que les choix fondamentaux.

Le chapitre 2 présente une analyse des quatre concepts sur lesquels s'articule notre étude. Le premier est celui *d'environnement de programmation* que nous regardons selon trois axes : les fonctionnalités, l'ergonomie et les relations avec une méthode de programmation. Le second est celui des méthodes de de programmation. Nous insistons sur le guide effectif qu'elle doit fournir. Le troisième est celui d'assistance au programmeur pour lequel nous étudions les formes qu'elle peut revêtir. Le quatrième est celui des relations entre l'environnement et le programmeur.

Le chapitre 3 présente les choix fondamentaux de Maiday. Le premier est l'intégration de Maiday au courant d'étude Nancéen sur la programmation. Notre ambition est de fournir un outil utilisable comme une *plate-forme expérimentale* dans ce domaine. Nous sommes donc naturellement amené à la méthode et au langage algorithmique Nancéen. Le second choix est d'utiliser intensivement des outils sophistiqués, un éditeur syntaxique en particulier, pour développer la maquette. Le troisième choix est la recherche constante de la simplicité (d'utilisation, d'apprentissage, de réalisation, ...) des outils développés.

1.3.2 Second volet : la Réalisation

Cette partie présente la réalisation et la validation de Maiday. De part la nature même d'une maquette, les fonctionnalités, l'architecture et le code de Maiday sont appelés à évoluer. Nous décrivons donc les choix de réalisation plutôt que les détails techniques de l'implantation. Le lecteur intéressé trouvera dans les annexes l'intégralité des programmes d'interface.

Le chapitre 4 est consacré à **EDME**, l'éditeur d'algorithme MEDEE. EDME possède quatre caractéristiques principales :

- ses commandes sont organisées selon une structure proche de la structure logique des algorithmes édités.
- il est réalisé par une couche logicielle enveloppant un noyau formé d'une éditeur syntaxique obtenu par une génération automatique. Les *méta-éditeurs* syntaxiques étant désormais diffusés, il nous a paru intéressant de proposer quelques critères de choix d'un tel outil à partir de notre expérience.
- ses commandes et fonctions sont implantées de façon à permettre l'ajoût ou la modification de fonctionnalités d'une façon relativement aisée.
- son interface utilisateur, utilisant des techniques classiques, a été très soigneusement étudié.

Le chapitre 5 traite du problème de la validation d'EDME. Il est en effet important que nous puissions évaluer la qualité de notre maquette et donc la pertinence de nos choix sur deux plans : celui de la réalisation et celui de l'utilisation. L'évaluation du premier point est effectuée par le biais du transport d'EDME sur une autre machine. Le second point est évalué par l'observation d'utilisateurs d'EDME et l'analyse de leurs critiques.

Le chapitre 6 présente un interprète d'algorithmes destiné à être intégré à EDME. Cet outil se révèle indispensable car, si EDME garantit la correction *définitionnelle* d'un algorithme, seul un mécanisme d'exécution peut en garantir la correction *opérationnelle*. Nous décrivons l'interprète, non encore implanté, sous la forme d'algorithmes MEDEE. Cette définition *méta-circulaire* a pour but d'étudier les extensions à apporter au langage et à EDME pour faciliter le développement de logiciels complexes.

1.3.3 Troisième volet : la Critique

Cette partie présente la critique de notre maquette. Elle comporte deux phases : l'analyse des problèmes perçus lors de l'utilisation d'EDME et la situation de notre travail par rapport aux systèmes développés par ailleurs.

Le chapitre 7 expose les problèmes majeurs mis en évidence par les premiers utilisateurs. Pour chacun des points évoqués (qualité de la présentation, gestion des messages, gestions des erreurs, inférences sur le texte, stratégies de construction) nous essayons de proposer des solutions remédiant aux difficultés. Nous essayons aussi d'en déduire les fonctionnalités nouvelles à adjoindre à la maquette actuelle. Il est important de noter que les critiques (parfois sévères) ne remettent pas fondamentalement en cause notre conception ni notre réalisation mais indiquent les extensions nécessaires que devra intégrer un véritable éditeur d'algorithmes.

Le chapitre 8 situe EDME par rapport à quatre environnements de programmation : MENTOR, le Programmer's Apprentice, CATY et bVlisp

Chapitre 2

La Programmation Interactive

2.1 Les environnements de programmation

La notion d'environnement de programmation est actuellement essentielle dans la branche du génie logiciel qui développe les outils qu'utilisera le programmeur de demain. Si depuis quelques années des outils spécifiques à la programmation existent, comme les éditeurs *pleine page*, les générateurs de programmes, les dictionnaires de données ou les vérificateurs de programmes, il manque souvent ce qui fait l'intérêt d'un environnement: l'intégration de ces logiciels dans un cadre unique. Par intégration, nous entendons ici plus que la simple inter-connexion d'outils comme on peut la trouver sous UNIX, INTERLISP ou Gandalf par exemple. Il faut que les différentes tâches du programmeur (analyse, codage, mise au point, documentation, preuve, ...) soient traitées d'une façon uniforme, tant en ce qui concerne les outils et les moyens d'interaction avec le poste de travail qu'en ce qui a trait aux concepts manipulés et aux *méthodes* de travail. Pour cela nous situons les environnements de programmation dans un cadre tridimensionnel:

l'axe des fonctionnalités. C'est ce que l'on appelle couramment la *boite à outils* du programmeur. Elle se compose d'un ensemble de logiciels permettant l'écriture, la validation, la compilation (si nécessaire), la documentation et la mise au point des programmes. De la richesse de cette boite va dépendre la puissance de l'environnement. Nous pouvons lui adjoindre aussi des facilités pour gérer les versions de systèmes, utiles essentiellement lors de la maintenance, et des outils de gestion de projet et de communications entre programmeurs, indispensables pour développer de grands projets. Nous caractérisons cet axe essentiellement par le langage de programmation pour lequel l'environnement est bâti et par la représentation interne des programmes, utilisée pour la communication entre les outils. Cette représentation peut être plate comme dans le système UNIX (fichiers structurés uniquement en caractères), arborescente comme dans Gandalf (MED,82) qui manipule des arbres abstraits ou encore spécifique à un langage comme les S-expressions dans les systèmes LISP (INTERLISP (TEI,78)) ou bVlisp (WER,84)).

l'axe ergonomique. Cet axe concerne la manière dont le programmeur communique avec l'environnement. Nous y plaçons certes les supports physiques de la communication (terminal alphanumérique, écran graphique et souris, ...) mais aussi l'architecture générale de l'atelier qui influe sur la structuration logique des outils telle que la perçoit le programmeur. Cet aspect est important car il conditionne en partie les méthodes de travail à adopter. Ainsi avec UNIX, en utilisant C, le cycle habituel pour créer un programme est d'écrire ce programme en entier, de le faire vérifier par **lint** (logiciel qui vérifie la syntaxe, la sémantique statique et donne divers renseignements sur les flôts de données) puis de le tester après compilation; avec bVlisp la création d'un programme sera plutôt un ensemble de modifications de fonctions LISP suivies immédiatement de leur test. Pour être efficace il faut dans le premier cas une démarche très planifiée, alors que dans le second une démarche par *essai-erreur* est possible. Il ne faut enfin pas perdre de vue

qu'un outil, aussi puissant et utile soit-il ne sera pas ou peu employé si son utilisation est trop complexe (cf la critique d'INTERLISP dans (G&W,84)). La complexité dans un environnement tient surtout à la multiplication d'outils différents, et une manière de s'en affranchir est d'uniformiser les interfaces en rendant transparent le changement d'outil, du moins sur le plan des commandes (mnémoniques, syntaxe et *effets*).

l'axe méthodologique. Les développements selon les deux axes précédents visent à repousser les limitations de la programmation dues aux logiciels (temps de réponses importants, difficultés d'inter-connexion, complexité d'emploi, ...). La limitation suivante est due au programmeur lui-même pour lequel la maîtrise d'un programme trop complexe n'est plus possible (cf T. Winnograd et la notion de *barrière de complexité*). L'aide que peut alors apporter un environnement de programmation est une incitation à suivre une méthode précise. La section suivante précise ce que recouvre pour nous le terme de méthode et pourquoi nous la jugeons indispensable. Adopter une méthode n'est profitable que si elle est rigoureusement appliquée et suivie, ce qui impose de fortes contraintes au programmeur. C'est pourquoi nous pensons qu'il est essentiel aujourd'hui d'intégrer la composante méthodique au niveau même des outils. Cette intégration doit inciter le programmeur à la rigueur sans qu'elle lui apparaîsse comme un fardeau ou lui supprime sa liberté de conception.

2.2 La méthode

L'informatique actuelle se trouve confrontée à un double défi: accroître le volume des programmes et leur qualité. Par volume, nous entendons aussi bien la taille de chaque application que le nombre total de celles-ci. Quant à la qualité, elle concerne la certification, l'absence d'erreurs, la documentation ou encore la facilité de maintenance. Ces deux problèmes sont liés si on considère l'importance économique de la programmation. Pour relever ces défis, plusieurs approches complémentaires sont développées, partant chacune d'un point de vue différent sur la programmation. Nous citons ici quelques-unes des approches possibles:

Une approche concerne les outils qui sont de plus en plus puissants (éditeurs structurels, metteurs au point symboliques, ...) et généralement intégrés dans un environnement cohérent. Citons par exemple Gandalf (MED,82), ADELE (EST,83), le Cornell Program Synthesizer (TEI,81), CONCERTO (AND,84) ou encore MENTOR (DON,84) et bVlisp (WER,84). Ces outils sont destinés essentiellement au développement de "gros" logiciels en *industrialisant* le poste de travail.

Une autre approche vise à automatiser la production de logiciel soit en synthétisant directement des programmes à partir de *spécifications* (citons PSI (BAR,79) ou SAFE (BAL,77) par exemple) soit, plus modestement par paramétrage de progiciels. On trouvera une présentation de ces approches dans (LAM,81) et (LAM,83).

Un autre domaine d'étude concerne les langages de programmation, soit très généraux comme Ada (ADA,80), soit plus spécifiques à un type d'application comme MAPPER (HER,84). Cette approche vise à donner les moyens d'exprimer plus efficacement les traitements ou les données. Elle est fortement inspirée des études sur les outils formels et conceptuels adaptés à la programmation comme les Types Abstraits de Données ou les descriptions de modèles de base de données. Il est également possible de situer dans ce domaine les travaux visant à offrir une vision nouvelle de la programmation comme les langages *orientés objets* (SMALLTAKL (GOL,80) par exemple) ou la *programmation logique* (PROLOG (COL,83) par exemple).

Nous pensons que parallèlement à ces voies, il importe d'étudier la rationalisation du processus de construction des programmes, c'est à dire en définir un modèle dont on peut dériver des lois et des méthodes effectives. Des travaux théoriques ont permis de fixer quelques méthodes.

P.C. Scholl (SCH,79) propose de ramener les données des programmes à deux structures fondamentales: la liste et l'arbre. Cette opération réalisée, il est possible de construire le programme en choisissant un schéma d'algorithme approprié au traitement de la structure des données et en l'instanciant, par des actions spécifiques au problème.

G. Guiho, C. Gresse et M. Bidoit (GUI,80) proposent une méthode de construction de programmes fondée sur les types abstraits auxquels sont associés des schémas de décomposition considérés comme la *structure algorithmique* du type. Cette méthode est présentée au chapitre 9 section 3.

C. Pair (PAI,79) propose la *méthode déductive* que nous présentons au chapitre 3 section 1.3.

C.Rich (RIC,78) propose d'adapter des méthodes d'ingénierie provenant de domaines divers, en particulier de l'électronique, à la programmation. Nous présentons succinctement la méthode développée au chapitre 9 section 2.

Les résultats théoriques ont donné naissance à des outils intégrant des méthodes de programmation. Citons par exemple CATY (GRE,84), le Programmer's Apprentice (RIC,81) (WAT,78) ou Maiday (GUY,84).

La liste des travaux que nous venons de citer n'est bien évidemment pas exhaustive.

L'étude que nous menons est limitée à la programmation et n'aborde pas la phase de spécification qui est souvent présentée comme nécessaire dans un développement rationnel d'applications. Nous pensons qu'en fait, une spécification n'est obligatoire que pour de gros programmes, elle l'est moins pour de petits problèmes dont l'énoncé est souvent assez précis. D'autre part, il est vraisemblable que de plus en plus d'applications de petites tailles et très personnalisées seront programmées par des utilisateurs possédant un poste de travail autonome. Ces programmeurs occasionnels auront certainement moins besoin de méthodes de spécification que de méthodes de programmation. Enfin, la programmation est une activité qui a déjà une histoire où interviennent des outils conceptuels, des critères de jugements et une grande expérience,

contrairement à la spécification qui est une activité encore jeune. Il est dès lors plus facile d'étudier des techniques, de réaliser et de valider des outils dans le domaine de la programmation.

La voie que nous proposons de suivre consiste à définir tout d'abord un *cadre méthodologique* à l'intérieur duquel seront construits les outils. Par cadre méthodologique nous entendons:

1. un modèle de l'activité de programmation, c'est à dire une description *formalisée* des différentes étapes du processus et de leur enchaînement.
2. un fil conducteur permettant au programmeur de toujours savoir «quoi et comment faire»
3. des règles précises guidant et justifiant les décisions.

Modéliser la programmation est une activité complexe qui, à notre connaissance, n'a pas encore abouti à des résultats universellement acceptés. Si certains points sont bien acquis comme par exemple l'intérêt de travailler par raffinements successifs ou de limiter le nombre des concepts (tant en ce qui concerne les données que les traitements), d'autres sont en revanche incertains. Ainsi ne connait-on pas les étapes nécessaires: doit-on passer par une phase de spécification formelle ? Doit-on mettre en équation le problème avant de programmer une solution ou peut-on programmer directement en adoptant une stratégie "d'essai-erreurs" ? Doit-on s'intéresser plûtot aux traitements ou plûtot aux données ? D'autre part, le domaine d'application choisi interfère aussi sur la programmation. Ainsi la *méthode Warnier* (WAR,79), en réduisant la programmation à l'analyse et à la description détaillée des fichiers d'entrée et de sortie s'applique bien dans les cas où d'importantes quantités de données sont manipulées avec des algorithmes simples. A l'opposé, lorsque le problème est algorithmiquement complexe, une méthode, telle que celle de E.W. Dijsktra (DIJ,76), qui accorde une grande place à l'analyse mathématique, est plus indiquée. La question du *fil d'Ariane* à fournir au programmeur nous semble cruciale. En effet, l'aide majeure que peut apporter une méthode tient à ce fil conducteur qui doit guider le programmeur, lui fournir des critères pour décomposer son problème et le résoudre et lui fournir un soutien pour dépasser les situations bloquantes, fréquentes en programmation. Il est là encore difficile de trouver un bon guide qui ne soit pas pénalisant tout en restant général. Il faut pour cela étudier et comprendre les différentes stratégies utilisées par les programmeurs confrontés à des problèmes particuliers et les intégrer dans un modèle de programmation cohérent. Il est surprenant que les méthodes proposées actuellement ne fournissent pas de tels guides; ainsi, par exemple, C.B. Jones (JON,82) propose un cadre élégant pour développer des programmes en un *continuum* allant de la spécification au code tout en prouvant à chaque étape l'adéquation des raffinements. Malheureusement il ne donne pas de fil conducteur mais seulement des conseils qui, pour utiles qu'ils soient, n'en fournissent pas pour autant une aide effective au programmeur.

La compréhension de programmes est une activité permanente pour les programmeurs; tant lors de la mise au point d'un nouveau code que lors de la maintenance de logiciels existants. Comprendre un programme revient à déterminer la fonction des variables employées, les structures de données qu'elles implantent, les algorithmes utilisés, ... C'est aussi essayer de deviner pourquoi telle implantation ou pourquoi tel style de programmation, par exemple. En fait il s'agit d'une reconstruction partielle des décisions et de leur

enchaînement réalisés pendant la conception du programme. Un programme peut donc être vu plus comme un *arbre de décisions* que comme un morceau de code. Cette conception nouvelle des programmes est discutée par D. Scott et W. Scherlis (SCH,83) et dans le groupe de travail ANNAGRAMM de l'AFCET. Il nous semble donc important que la méthodologie suivie permette d'expliciter et d'enregistrer les différents choix ainsi que leur enchaînement.

Les travaux présentés dans cette thèse n'ont pas pour but de définir une méthode générale mais d'étudier et d'expérimenter des outils en se situant dans un cadre méthodologique précis.

2.3 Les outils et l'assistance au programmeur

Une des difficultés de la programmation concerne le grand nombre d'informations disparates et de concepts disjoints qu'il est nécessaire de manipuler simultanément. Ainsi, lors de l'écriture de code il faut avoir à l'esprit l'algorithme et les structures de données à implanter, la syntaxe et la sémantique du langage de programmation utilisé, les règles permettant de transformer l'algorithme en programme, les commandes de l'éditeur que l'on utilise. En général c'est dans ce même temps qu'on essaye d'évaluer les performances, la qualité du code, ... Lors de l'exécution d'un jeu d'essai, il faut se représenter l'algorithme, son implantation, avoir une image du fonctionnement de la machine, faire se correspondre les valeurs réelles (correspondant au problème), souhaitées (correspondant au test) et obtenues (correspondant à la réalité du calcul); conjointement, il faut envisager, confronter, valider, réfuter toutes les hypothèses susceptibles d'expliquer les dysfonctionnements du programme. Une autre difficulté apparaît dès lors que les applications grossissent car il faut alors situer dans la vue macroscopique de l'application (son architecture, sa décomposition, ...) le code travaillé dont on n'a qu'une vue microscopique (l'implantation et tous ses détails).

Il apparaît donc que l'un des buts essentiels d'un environnement de programmation doit être de diminuer cette complexité en déchargeant le programmeur de ce qui ne concerne pas directement son activité du moment.

Une autre caractéristique de la programmation est la rapidité et la fréquence des changements d'activité. Ainsi on souhaite tester un morceau de code sitôt qu'il est écrit, ou lire un fragment de programme pendant qu'on en édite un autre. Là encore, un environnement doit faciliter cette *versatilité*.

Partant de ces constatations nous voyons cinq formes possibles à l'assistance au programmeur.

2.3.1 La documentation en ligne

Cette première forme concerne les systèmes d'aide que le programmeur appelle explicitement. Ce qui est demandé est ici un renseignement précis, très ponctuel. Par exemple, quelle commande faut-il activer pour réaliser telle action ? Comment appelle-t-on l'outil qui permet de formater un texte ? Quel est le type de cette variable ? ... Ce type d'assistance est maintenant courant sur les systèmes diffusés (fonction **apropos** sur EMACS, **HELP** sur MENTOR, **man** sur UNIX ou **help** sur MULTICS). Ces systèmes d'aide nous semblent absolument essentiels car ils libèrent la mémoire du programmeur (seul ce qui est relatif à l'action en cours est à retenir) tout en évitant les blocages dus à un oubli ou une connaissance incomplète. La conception de ces systèmes doit intégrer des objectifs divers et quelquefois contradictoires. En effet, les réponses doivent être brèves, précises et pertinentes mais les questions doivent pouvoir être *floues*, quitte à être *affinées* avec l'aide du système. Il est aussi important que dans un environnement, les aides se présentent sous des formes cohérentes et si possible identiques.

2.3.2 La préparation de la tâche suivante

Cette seconde forme a trait à des systèmes qui travaillent parallèlement au progammeur, d'une façon invisible, pour préparer d'autres activités. Les activités à préparer sont multiples et vont de l'utilisation d'autres outils de l'environnement (documentation des fonctions en bVlisp, interprétation dans le Cornell Program Synthesizer dont les données sont construites pendant l'édition même) à l'analyse d'une session de travail (Listes historiques de commandes, ou système **audit** de MULTICS). L'aide provient ici de la rapidité avec laquelle il est possible de passer d'une tâche à une autre, ainsi que de la transparence des interfaces entre outils.

2.3.3 Les contrôles interactifs

Cette troisième forme vise les instruments de contrôle des textes. Une activité très prenante et fastidieuse est la recherche dans un texte des erreurs de syntaxe et de sémantique statique (contrôle de types, nombre de paramètres des fonctions, ...). Un remède nous semble être de vérifier le texte et d'informer le programmeur le plus rapidement possible des erreurs.

La rapidité est intéressante à deux points de vue: d'une part elle accroît le *confort* (plus de recherche fastidieuse dans un listage de compilation, plus besoin de se demander pour chaque instruction frappée si

elle est correcte) et d'autre part, elle augmente la concentration sur l'aspect intelligent qu'est la *compréhension* de ce que réalise le programme: la machine joue un rôle très efficace de *garde-fou*.

2.3.4 La prise en charge d'activités secondaires

Cette quatrième forme concerne l'automatisation de certaines tâches secondaires vis à vis de l'activité en cours. Les traitements automatiques nous semblent intéressants moins parce qu'ils suppriment totalement un travail que parce qu'ils permettent de séparer nettement les activités tout en assurant un passage rapide de l'une à l'autre. Par exemple, un formateur de programmes est très utile en séparant les fonctions d'écriture et de lecture des programmes, mais le formatage définitif pourra être fait à la main pour accroître la lisibilité ou se conformer à des normes différentes de celles du formateur. D'autres types d'automatisations concernent la génération et la modification des textes, par exemple, les fonctions **trace** et **untrace** des systèmes LISP qui effectuent des modifications pré-définies pour faciliter l'étude du flôt de contrôle à l'exécution. Il est possible de mettre dans ce type d'assistance les outils de gestion de versions et d'archivage (comme dans bVLISP, Gandalf ou la base de programmes d'ADELE (EST,84)).

2.3.5 Le pilotage de l'utilisateur

Dans cette cinquième forme d'assistance, c'est l'outil qui prend en charge un processus complexe. Le programmeur est alors questionné sur ce que le système ne peut construire seul. Dans le cadre de la programmation de tels outils pourraient être, par exemple, des démonstrateurs de programmes, l'utilisateur ayant à fournir les propriétés spécifiques ou les théorèmes hors de portée du démonstrateur, l'outil construisant seul la preuve.

2.4 Les relations entre l'outil et le programmeur

Lors de la conception d'un environnement de programmation, il faut tenir compte d'une composante nouvelle: l'homme. En effet, la qualité de l'interaction entre l'environnement et le programmeur est un facteur important qui fait accepter ou rejeter l'outil.

Dans cette section nous identifions les différentes composantes ergonomiques d'un environnement de programmation en essayant, pour chacune d'elle, de définir les types de problèmes posés et les solutions envisageables. Nous avons isolé quatre composantes: une composante matérielle qui concerne les moyens

physiques d'interaction, une composante conceptuelle qui concerne la perception des objets manipulés par l'environnement, une composante qui concerne les méthodes de travail et une composante psychologique.

2.4.1 La composante ergonomique

L'aspect ergonomique est essentiellement la prise en compte de la technologie du poste de travail. Une bonne adaptation de l'interface utilisateur aux possibilités du matériel est importante pour ce qui concerne le confort. En effet, chaque appareil possède ses contraintes propres qui doivent être étudiées spécifiquement. Ainsi, si l'on décide que le programmeur interagira avec le système à travers un terminal alphanumérique classique, il convient d'etre conscient que des limitations importantes existent: le clavier est un instrument d'un emploi délicat, il faut alors prévoir des procédures pour rattraper les erreurs ou diminuer la quantité de texte à frapper, l'écran pour sa part est de taille très restreinte (le tiers d'une feuille de format courant) et se révèle pénible à lire attentivement, il faut donc bien isoler les points de l'écran qui sont essentiels, organiser l'information sur un espace réduit, ... Si on emploie un terminal à point (*bit-map*) et une souris, il faut veiller à minimiser les déplacements de la souris et maintenir une liaison univoque entre le curseur et la structure interne sur laquelle il pointe; ce problème est non trivial dans le cas d'un éditeur syntaxique par exemple.

2.4.2 La composante conceptuelle

La perception des outils et des données est très importante car elle structure totalement ceux-ci dans l'esprit du programmeur. La contrainte fondamentale ici est d'offrir une vision unique qui n'oblige pas l'utilisateur à maintenir la cohérence entre deux structures trop différentes pour le même objet. Il est important aussi que les représentations externes ne soient pas trop réductrices, par exemple un système graphique de représentation des programmes fondés sur des organigrammes risquerait de trop focaliser l'attention sur les flots de contrôle au détriment de la structure du programme.

2.4.3 La composante d'organisation du travail

Nous pensons que les méthodes de travail du programmeur sont liées aux relations qui existent entre les outils et lui. Certes beaucoup d'autres facteurs existent comme le langage utilisé qui, selon qu'il est interprété ou compilé, induira des méthodes différentes de développement de programmes. De même la puissance des outils dont on dispose influe largement. Néanmoins l'organisation de l'environnement telle

qu'elle est perçue joue un rôle. Ainsi, par exemple, dans l'environnement C sous UNIX, les outils sont disjoints et organisés en *chaînes*:

éditeur -> lint -> compilateur C -> éditeur de liens -> *debugger*

ce qui instaure un cycle de développement à longues phases, à l'opposé de bVlisp où tous les outils sont sur le même plan, accessibles à tout moment: le cycle de développement est composé de multiples cycles (un par fonction) à phases très brèves. Dans ces deux exemples, aucune contrainte sur les cycles n'est explicitement imposée, le choix de la longueur de cycle est très fortement influencé par des considérations sur le temps de chargement des outils, la longueur des *temps morts* par rapport aux *temps actifs* (temps de compilation et temps d'exécution du jeu d'essai par exemple) ou encore la facilité de *changer de contexte* de travail (nouvelles commandes, syntaxes différentes, ...).

2.4.4 La composante psychologique

Toutes les recherches et les tentatives pour industrialiser ou rationaliser la programmation aboutissent toujours à contraindre le programmeur. Ces contraintes vont de la définition de normes de présentation des programmes (indentations, typographie, documentation, ...) à l'emploi de styles de programmation (par types abstraits, par objets, ...) en passant par le suivi de démarches et d'étapes. Ainsi la méthode déductive que nous employons oblige le programmeur à écrire un algorithme avec une présentation rigoureuse (les trois colonnes) puis à coder en employant des règles strictes pour transformer l'algorithme. Ceci est en contradiction avec l'opinion des programmeurs qui considèrent leur activité plutôt comme un *art* (WEI,71). Il est donc essentiel que les outils proposés intègrent ces contraintes sans rebuter l'utilisateur. Pour cela nous estimons qu'il faut penser ces outils en terme de coopération avec le programmeur.

La coopération peut se manifester selon trois formes:

1. Le programmeur doit pouvoir se décharger sur l'outil des tâches fastidieuses. C'est l'outil qui fait les contrôles, entretient les cohérences nécessaires, et signale au programmeur ses oublis.
2. L'outil doit proposer des procédures de récupération d'erreurs souples et adéquates, il devrait même les *suggérer*.
3. Le programmeur doit avoir la possibilité de dialoguer avec l'outil, de l'interroger à tout moment sur son état, sur l'état du travail, sur les influences prévisibles de telle action, ... Symétriquement, le système devrait pouvoir interroger le programmeur.

Les techniques à mettre en oeuvre dans de tels systèmes relèvent pour beaucoup de l'intelligence artificielle et nécessitent encore de nombreux développements, mais des outils partant de ces principes commencent à apparaître, citons le Programmer's Apprentice (RIC,81) ou bVlisp (WER,84).

Chapitre 3

Une approche: Maiday

Dans le chapitre précédent, nous avons présenté les qualités que nous attendons d'un environnement de programmation, en particulier sur les plans de l'assistance au programmeur, de l'ergonomie et des méthodes de programmation ainsi que les problèmes qu'elles soulèvent. Cette analyse a fortement inspiré la conception de Maiday, l'environnement de programmation que nous développons. Nous présentons ici un cahier des charges de Maiday, en insistant particulièrement sur les choix fondamentaux de conception.

Nous présentons tout d'abord les objectifs et les motivations du projet. Une des motivations principales de Maiday est d'obtenir une *base expérimentale*. Celle-ci doit permettre, dans un premier temps, de valider les idées actuelles, puis, par la suite, nous espérons que Maiday servira de support pour l'étude d'outils et de techniques actuellement en cours d'élaboration. Cette motivation a eu une certaine influence sur l'architecture générale de Maiday que nous présentons ici. Un second facteur influant sur la conception de Maiday a été le cadre méthodologique que nous avons retenu. Nous présentons donc dans ce chapitre la *méthode déductive* que nous avons utilisée et son langage support: MEDEE.

Nous nous sommes essentiellement consacré à l'étude du processus de création d'un algorithme et l'outil développé, EDME, peut être considéré comme le coeur de Maiday. Nous présentons donc ici aussi un cahier des charges d'EDME. Nous insistons en particulier sur les aspects de structuration du texte et des commandes, sur l'aspect des contrôles et de leur intégration au commandes et sur l'aspect ergonomique.

3.1 Le projet général

3.1.1 Les objectifs de Maiday

Depuis 1974, une recherche active est menée à Nancy sur la programmation, en particulier sur l'aspect méthodologique. Nous avons déjà montré les grandes lignes de la méthode développée et du modèle de programmation qui lui est associé (PAI,79). En relation avec cet aspect méthodique divers outils et techniques sont étudiés.

En ce qui concerne les outils, l'effort a essentiellement porté sur le passage d'un algorithme à un programme. B. Huc (HUC,78) a écrit un compilateur qui traduit un algorithme écrit en MEDEE en un programme PASCAL. L'intérêt de ce travail a été de montrer qu'un outil automatique pouvait prendre en

charge les contraintes d'exécution, en particulier l'ordre d'évaluation des variables qui est différent de leur ordre de définition dans l'algorithme. Une étude a également été entreprise (COY,82) pour construire un éditeur de programmes PASCAL qui utilise la méthode déductive. En utilisant MENTOR comme support de développement, ce projet a principalement apporté une expertise dans l'utilisation d'outils modernes.

Pour ce qui concerne les techniques, les travaux ont pour objectif d'améliorer et d'optimiser les algorithmes. En effet, le respect de la démarche déductive conduit souvent à introduire des intermédiaires commodes pour l'algorithme, mais inutiles dans le programme. Lorsque ces intermédiaires sont des suites, il devient crucial de les éliminer pour conserver une certaine efficacité. J. Souquieres (SOU,82) (FIN,84a), a développé un système fondé sur des transformations d'objets du type abstrait suite. F. Bellegarde (BEL,84) étudie des techniques d'optimisation en utilisant un langage fonctionnel et un système de réécriture.

Des travaux portent aussi sur l'amont de la programmation: les spécifications. En reprenant les idées de la méthode déductive, J.P. Finance (FIN,79) a donné un cadre formel et méthodologique à la spécification. E. Dubois (DUB,84) a pour sa part traité deux cas concrets de système d'information, ce qui lui a permis de définir un langage et un *méta-algorithme* adapté à la spécification de gros systèmes d'information. Enfin, l'étude et la réalisation d'un environnement de spécification (BUY,83) (FIN,84b) sont actuellement en cours. L'objectif principal de cet environnement est de permettre l'édition plus ou moins assistée de spécifications, l'édition doit ici être comprise comme la création et les modifications habituelles sur des textes, mais aussi comme la possibilité de transformer ceux-ci automatiquement (*concrétisation* de types abstraits, explicitation de définitions, ...).

Enfin, comme toute idée nouvelle, la méthode déductive requiert une évaluation qui doit permettre de déterminer sa pertinence, son domaine d'application, son apport tel qu'il est perçu par les utilisateurs. Quelques études de ce type ont été entreprises dans le cadre d'observation de programmeurs par des psychologues (KOL,79). Actuellement, une étude menée par J.M. Hoc (HOC,84) est centrée sur la mise en évidence des stratégies qu'emploient les programmeurs lors de la création d'algorithmes.

Une autre forme d'évaluation de la méthode et du langage MEDEE, est leur utilisation dans l'enseignement. Depuis quelques années, ils sont employés dans les universités et à l'IUT de Nancy comme support des cours d'initiation à l'algorithmique et à la programmation. De cette expérience, vécue en tant qu'élève et qu'enseignant, nous pouvons tirer quelques conclusions:

> L'écriture d'un algorithme est perçue comme étant plus difficile que l'écriture directe du programme. A ceci, nous voyons au moins deux explications. Les exercices sont de petite taille et le programme est souvent trivial. Mais surtout, l'absence de moyen d'édition des algorithmes oblige à réécrire beaucoup de texte (c'est la contre-partie de l'intégration de la documentation à l'algorithme).
>
> Comme MEDEE n'est pas doté d'un environnement d'exécution, en particulier nous ne possédons pas de compilateur, les algorithmes doivent être traduits dans un langage de programmation (PASCAL en général) pour être mis au point. Il est donc nécessaire de manipuler deux langages simultanément qui sont différents dans l'esprit (MEDEE est statique, définitionnel et PASCAL est dynamique, opérationnel) mais qui restent proches d'un point de vue syntaxique (pour permettre un passage simple de l'algorithme au programme). Ceci a pour conséquence de doubler le volume d'apprentissage et d'obscurcir la perception des spécificités de chacun des langages.
>
> La mise au point de l'algorithme est rendue complexe par l'obligation de passer par un langage de programmation.
>
> Le programmeur a seul la responsabilité de contrôler son travail, ceci conduit inévitablement à des déviations plus ou moins heureuses de la syntaxe du formalisme ou de l'application de la méthode lorsqu'apparaît un problème. De ce fait le caractère rigoureux qui fait l'intérêt de la méthode est perdu.
>
> Il est possible de constater une amélioration de la programmation des étudiants au cours de l'apprentissage de la méthode, de plus il y a une certaine corrélation entre la qualité du code et l'assimilation de la méthode.

Les critiques exposées ci-dessus correspondent aux aspects les plus visibles, des aspects moins apparents comme la bonne structuration générale des programmes, la forte densité de commentaires dans le code ou la taille des problèmes abordés montrent les bénéfices que l'on tire de l'utilisation d'une méthode rigoureuse. Une réponse possible aux problèmes évoqués est de fournir aux étudiants des outils facilitant l'édition, la mise au point des algorithmes puis le passage à un programme efficace.

Les objectifs du projet Maiday sont dès lors clairs:

1. Les différents travaux sur le langage, les outils et les techniques sont disjoints. Il est important de les moderniser en les unifiant en un ensemble logiciel cohérent qui permettra une validation plus facile des idées.
2. Les travaux connexes comme les transformations de spécifications en algorithmes ou l'étude des stratégies de programmation ont besoin d'une *base expérimentale* sur laquelle bâtir les outils et les procédures.
3. Les outils d'enseignement de l'algorithmique et de la programmation sont en retard sur l'évolution du matériel. En particulier, l'apparition de postes de travail personnels puissants (les micro-ordinateurs 16 bits par exemple) permet d'envisager des outils efficaces et "intelligents", utilisables pour l'enseignement.
4. Les outils de génie logiciel deviennent de plus en plus sophistiqués et la programmation de systèmes importants s'effectue de plus en plus par le développement d'une couche logicielle supplémentaire sur un système de base. Il est important d'acquérir une expertise dans l'utilisation de tels outils et de participer à leur évaluation expérimentale.
5. Il est certainement prématuré de vouloir construire directement un environnement de programmation efficace et *diffusable*. Nous pensons qu'il est nécessaire d'expérimenter des outils, des techniques et des idées afin de pouvoir dégager de vraies spécifications pour un outil réellement opérationnel. De plus, nous espérons que cette expérimentation autorisera une extrapolation vers les outils de production industrielle de logiciel.

3.1.2 L'architecture générale de Maiday

Maiday est conçu comme un environnement de programmation: c'est à dire comme un ensemble d'outils différents qui utilisent et travaillent sur une structure de donnée commune. Comme l'essentiel des traitements est la modification de textes écrits dans un langage formel, la structure choisie est celle d'arbre abstrait. Ceci est bien sûr lié à l'outil de développement retenu qui est MENTOR. Nous verrons plus tard comment aujourd'hui nous envisagerions un tel choix; au moment du lancement du projet (printemps 82), seul MENTOR avec le générateur METAL était disponible, de plus l'équipe possédait déjà une certaine connaissance sur ce logiciel (GUY,80).

L'outil le plus attendu est un éditeur d'algorithmes, le travail a donc essentiellement porté sur cet aspect de l'environnement d'autant plus que nous le souhaitons le plus *intelligent* possible. L'éditeur est le coeur de Maiday.

Lorsqu'un algorithme a été écrit, il convient en général de le mettre au point. Pour cela, l'outil le plus efficace est un interprète qui permet *d'évaluer* un algorithme, et même un morceau d'algorithme, sitôt qu'il est édité. Pour utiliser ensuite l'algorithme, il est possible de le compiler en un programme PASCAL qui sera exploité classiquement avec de bonne performances.

Eventuellement, des outils permettant d'aboutir à un algorithme en partant d'une spécification, comme le système développé par L. Soufi (SOU,84) qui permet *d'expliciter* une spécification écrite en SPES en un algorithme MEDEE, ou ceux permettant *d'optimiser* un algorithme seront intégrés à Maiday.

La figure 2 présente le schéma général de Maiday.

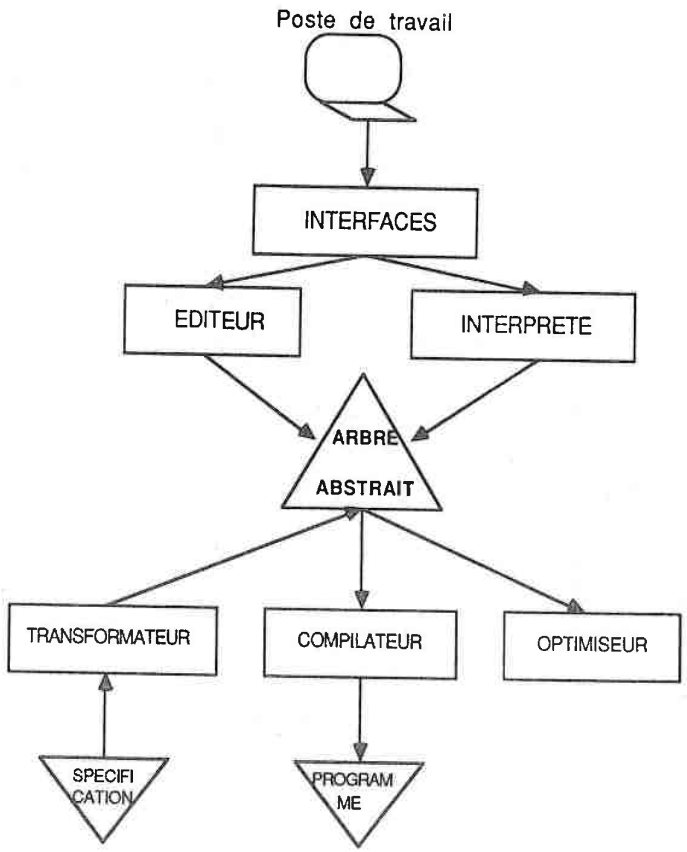


Figure 2. L'architecture générale de Maiday

3.1.3 La méthode implantée

La méthode retenue est la *méthode déductive*. Elle est développée au CRIN depuis 1974 et se situe dans le courant de la programmation structurée. Un langage support, **MEDEE**, a été défini (BEL,77) et amélioré (DUC,84). Les principes généraux et les idées sous-jacentes sont exprimées dans (PAI,79). Le modèle de l'activité de programmation retenu est résumé dans le schéma suivant:

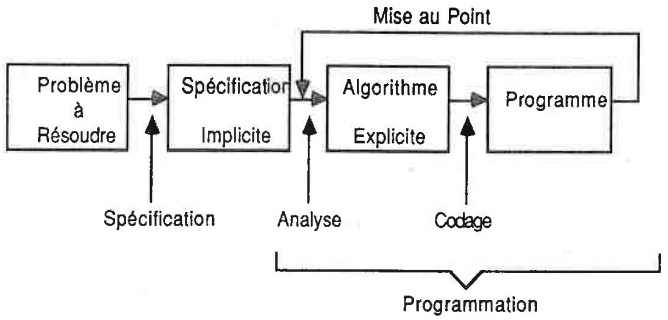


Figure 3. Modèle de l'activité de programmation

La première phase vise à définir formellement le problème, à le *mettre en équations* sans tenir compte de sa résolution (phase de spécification). On obtient alors des *définitions implicites* dans lesquelles sont utilisés des quantificateurs, des types de données abstraits, etc ... Partant de cette spécification, on peut alors *expliciter* le problème, c'est à dire en donner une résolution algorithmique qui soit *statique* et ne tienne pas compte des contraintes d'exécution (phase d'analyse). Ensuite on peut traduire l'algorithme en un programme par l'utilisation de règles de transformation systématiques de l'algorithme MEDEE en programme exécutable (phase de codage). Puis on effectue la mise au point classique. Le départ de ce modèle est dessiné séquentiellement mais il est bien évident que des retours en arrière sont possibles et même inévitables.

Nous ne nous prononçons pas ici sur la pertinence du modèle que seules des expérimentations permettrons d'évaluer dans le cadre d'environnements de programmation. D'autres modèles sont envisageables. Dans le PROGRAMMER'S APPRENTICE (RIC,81), le modèle met l'accent sur l'aspect des réutilisations de *plans* déjà construits; dans CATY (GRE,84) ou dans la méthode proposée par P.C. Scholl (SCH,79), le modèle

insiste beaucoup sur les données et leur structure couplée à des schémas de programmes. Les différentes phases sont aussi critiquables, en particulier dans (GUI,80) et (GRE,84), on trouve une critique de la phase de spécification. Ces auteurs considèrent les difficultés inhérentes à la spécification, par exemple: l'absence d'un "bon" langage de spécification, la complexité de la tâche du spécifieur qui nécessite des outils puissants, encore inexistants, pour être menée à bien, la difficulté de valider une spécification ou encore le passage d'une spécification formelle à un programme qui est un problème non résolu. Ils proposent donc une méthode permettant de construire directement des programmes.

L'expérience que nous menons se concentre essentiellement sur les phases d'analyse et de codage, les problèmes traités étant courts et ayant généralement des énoncés précis.

Le *fil d'Ariane* que nous suivons est celui de l'analyse déductive qui peut se résumer simplement en:

1. Ce que l'on connait le mieux en commençant, c'est le résultat souhaité. Il faut donc le définir formellement.
2. On explicite le résultat à l'aide d'une définition qui peut être:
   - simple (expression algébrique ou fonction auxiliaire (analyse récursive))
   - conditionnelle (analyse par cas)
   - itérative (analyse récurrente)
3. Dans la définition, de nouveaux objets ont certainement été introduits, on leur donne alors un commentaire (définition informelle)
4. On retourne en (1) en prenant pour résultat un des objets introduits. L'analyse s'arrête lorsque les objets introduits sont les données du problème.

Il est possible de noter quelques points:

Le processus est très adapté lorsqu'on cherche une solution sans a priori; il l'est moins si l'on veut réutiliser des objets ou algorithmes pré-existants.

La documentation est intégrée au processus de construction.

Les critères de décomposition sont peu nombreux et les choix bien visibles: ils se traduisent en une des trois formes de définitions formelles.

L'arbre des décisions n'est pas explicitement traité dans le cadre méthodologique où nous nous situons. Toutefois, lors de l'utilisation de définitions itératives et conditionnelles d'objets, l'utilisateur introduit fréquemment des modules qui sont des unités syntaxiques correspondant à la résolution d'un sous-problème. Les sous-problèmes typiques sont la définition du $I^{ème}$ terme d'une suite récurrente et de la valeur d'un objet sous une condition donnée. L'arbre des modules (ainsi que le commentaire obligatoire associé à l'apparition du nom du module) reflète un chemin précis dans l'arbre des décisions. Il manque encore un formalisme permettant d'exprimer aisément l'arbre des décisions et de l'intégrer à une démarche.

3.1.4 Le langage MEDEE

MEDEE est le langage support de la méthode déductive. Nous avons déjà cité certaines de ses caractéristiques, à savoir son aspect statique, modulaire et algorithmique. Il permet en outre de construire et d'exprimer un algorithme en utilisant la méthode.

Les concepts de bases sont au nombre de trois:

L'objet. Un objet est l'entité calculable. Il possède un type (grossièrement identique à un type statique de PASCAL) mais contrairement à une variable informatique, il ne peut recevoir qu'une seule valeur: MEDEE est un langage à affectation unique. Pour permettre d'exprimer les itérations, les objets font tous partie du type implicite **SUITE**. Le symbole spécial **@** permet l'accès au dernier élément défini de la suite.

La définition. Une définition est une équation qui indique comment est calculé un objet. MEDEE possède trois formes de définitions:

- la définition simple: soit l'objet est une donnée, soit il se calcule grâce à une expression algébrique ou une fonction auxiliaire le liant à d'autres objets.
- la définition conditionnelle: l'objet est défini par une analyse par cas.
- la définition itérative: l'objet est défini par approximations successives.

Le module. Un module est une unité syntaxique introduite lors de l'emploi des définitions structurées. Cette unité correspond généralement au traitement d'un sous-problème. Les sous-problèmes typiques sont la définition d'un terme d'une suite (module itératif) ou d'une valeur d'un objet sous une condition (module conditionnel).

MEDEE, de par l'aspect statique des définitions, l'unicité des affectations, l'emploi de la suite comme type fondamental et le cadre mathématique peut être rapproché de LUCID (ASH,77). Il s'en distingue par son objectif premier qui est de faciliter la construction des programmes plutôt que leur preuve. De ce fait, il possède une syntaxe inspirée des langages du type d'Algol qui permet une traduction simple de l'algorithme en un programme en utilisant des règles essentiellement syntaxiques.

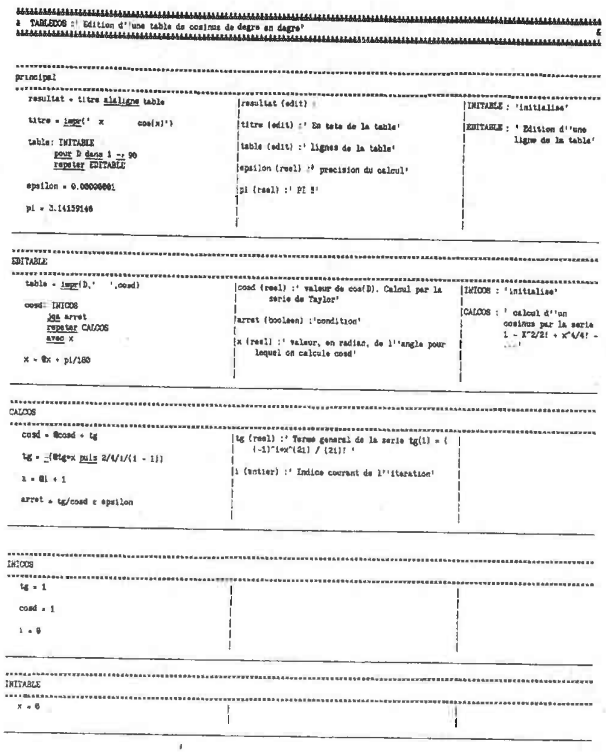


Figure 4. Un algorithme MEDEE

La syntaxe nous semble suffisamment "intuitive" pour que nous ne nous attardions pas à commenter ligne à ligne l'algorithme. Notons simplement quelques points importants:

Le texte est décomposé en trois parties, à gauche figurent les définitions formelles des objets, au centre nous trouvons le lexique, c'est à dire les définitions informelles des objets nouvellement introduits, à droite se situe le lexique des modules dans lequel sont commentés les modules introduits dans les définitions structurées. L'aspect *documentation* est très important. Il facilite évidemment la relecture et la compréhension d'un algorithme, mais surtout il fournit un support simple au guide méthodologique: tout objet introduit doit apparaître dans le lexique, il suffit ensuite de chercher dans le lexique un objet non formellement défini pour continuer l'algorithme. Cette procédure simple, appliquée systématiquement fournit un moyen sûr de vérifier si la construction d'un algorithme est achevée.

Les suites définies peuvent comporter plusieurs niveaux, ceci est indiqué par leur *profondeur itérative*, c'est à dire le nombre de modules itérés dans lesquelles elles sont définies. Dans l'exemple, la profondeur maximale est de 2.

Une définition itérative définit le dernier terme de la suite. Le terme courant est calculé dans le module qui apparaît en seconde position, le premier module est utilisé pour définir la valeur initiale.

La syntaxe de MEDEE n'est pas encore définitivement figée, l'implantation de Maiday utilise une forme ancienne, bien connue, pour laquelle des outils sont récupérables en particulier un module de traduction MEDEE−PASCAL. Le lecteur intéressé trouvera dans (DUC,84) un cours sur la méthode déductive utilisant un MEDEE légèrement différent. Il est clair que le développement de Maiday et surtout son utilisation donneront des éléments pour améliorer l'aspect syntaxique de MEDEE en particulier sur le plan de la compréhension intuitive des définitions.

3.2 L'édition des algorithmes

3.2.1 Les objectifs généraux

Dans le chapitre précédent, nous avons présenté les domaines dans lesquels se situe notre étude. Les objectifs d'EDME, EDITEUR pour MEDEE, s'en déduisent aisément; ils reprennent aussi les buts du projet Maiday. Nous voudrions ici présenter EDME selon deux points de vue très différents: celui de l'utilisateur et celui du réalisateur. Nous pensons que la conception d'outils destinés à des utilisateurs non-informaticiens, comme EDME, doit être envisagée comme un compromis entre ces deux visions. En effet, les nombreuses facilités, le caractère agréable de l'interaction, son côté "esthétique", la brièveté des temps de réponses que réclame l'utilisateur ne doivent pas empêcher un code de qualité: clair, lisible, orthogonal dans ses

fonctions, ... De même le réalisateur doit toujours penser que l'utilisateur n'a pas l'habitude de manipuler les langages formels et les concepts de la programmation, que la représentation intellectuelle des structures manipulées sera probablement différente pour chacun et donc que telle commande, liée à la structure de donnée sous-jacente, semblera naturelle au spécialiste mais ne sera pas assimilée par le débutant.

3.2.1.1 Le point de vue de l'utilisateur

La qualité primordiale que nous voudrions développer se résume en un mot: **simplicité**.

Simplicité des concepts. Comme nous l'avons déjà affirmé, à chaque instant, l'utilisateur ne doit manipuler qu'un concept à la fois. Pour cela, il faut qu'ils soient en nombre limité avec des articulations simples et compréhensibles. En particulier, il convient de noter qu'il y a simultanément trois niveaux de représentations: un dans l'esprit de l'utilisateur, correspondant aux concepts syntaxiques et sémantiques du langage; un second dans l'image de la représentation interne que donnent les commandes; le dernier dans la visualisation du "texte". Il nous semble essentiel que ces trois niveaux aient une structuration identique et surtout qu'il existe des liens univoques entre chaque niveau. Un modèle de structure assez général pour résoudre ce problème est celui *d'ensembles emboités*, nous verrons dans la section suivante comment nous avons adapté et exploité ce modèle.

Simplicité des manipulations. La "manipulation" possède un double aspect: logique qui concerne l'action sur le texte et ergonomique qui concerne la façon d'activer les commandes.
L'aspect logique est caractérisé par les questions que peut se poser l'utilisateur: «Quelle commande activer pour effectuer telle action ?» «Quelles sont les commandes pertinentes en un point donné du texte ?» par exemple. Nous pensons qu'ici la simplicité provient de la structuration évoquée dans le paragraphe précédent. En effet, si un seul concept est manipulable à un instant donné, le nombre de commandes pertinentes est réduit et *l'effet attendu* (i.e. le résultat, les conséquences) de chacune est très précis. De plus, une organisation forte et rigide des commandes favorise l'apprentissage.
L'aspect ergonomique est certes lié au terminal utilisé mais aussi aux techniques de lancement de commandes offertes à l'utilisateur. Ces dernières doivent souvent concilier des caractéristiques difficilement compatibles comme la facilité d'emploi, pour les débutants, et l'efficacité, pour les experts du système. Il est certainement indispensable de pouvoir proposer plusieurs méthodes dont, par exemple, les menus, la frappe directe d'un nom explicite, les *clés* (cf EMACS), les touches de fonctions, les désignations directes (souris, déplacement curseur). Le problème est alors de permettre les changements à la volée de méthode, et même, éventuellement de proposer celle qui semble la mieux adaptée.

Simplicité de création d'algorithmes. Créer un algorithme est l'activité centrale pour laquelle EDME est conçu. C'est une activité complexe par elle-même et la simplicité provient de la prise en charge par EDME de trois éléments:

1. les contrôles et la rétro-action immédiate en cas d'erreur
2. la détermination et l'enchaînement des informations obligatoires mais non créatives (types d'objets, commentaires) à demander au programmeur

3. la gestion, la génération, voire l'inférence, des informations syntaxiques (en-tête de module par exemple) et contextuelles (calcul des listes "avec", des objets à initialiser, types d'objets par exemple).

Simplicité de communication. L'utilisation d'un outil interactif peut être vue comme deux flôts d'informations générés l'un par le système et l'autre par l'utilisateur. Si l'homme a la maîtrise du second, il faut absolument organiser le premier. L'information doit en effet être explicite, non ambiguë et triée. Nous pensons que des trois qualités évoquées, la dernière est à étudier plus particulièrement parce qu'elle influence beaucoup les deux premières et aussi parce qu'elle est plus à notre portée en tant que réalisateur.

Simplicité d'apprentissage. L'apprentissage est un moment privilégié des relations entre l'outil et l'utilisateur. S'il se révèle trop ardu, il y a des chances pour que l'outil ne soit pas utilisé; nous avons personnellement mis de l'ordre de deux mois pour apprendre à utiliser MENTOR et EMACS; seule une très forte conviction, a priori, que ces outils étaient indispensables, nous a fait persévérer. Afin de rendre EDME facile à apprendre, nous pensons qu'il faut développer trois qualités: premièrement, une rétro-action immédiate doit être associée à chaque action facilitant ainsi la compréhension des commandes, deuxièmement il faut sécuriser l'utilisateur en assurant qu'aucune erreur n'aura d'influence sur l'algorithme édité, et troisièmement il faut localiser précisement et *expliquer* clairement les erreurs commises. Cette politique peut s'exprimer par ce conseil:
«Essayez et regardez ce qui se passe !».
Elle n'est bien sûr pas incompatible avec l'existence de systèmes d'aides interactives et de guides d'apprentissages (les *tutorials*).

3.2.1.2 Le point de vue du réalisateur

EDME, tel qu'il est développé ici, est une maquette, c'est à dire qu'il doit répondre à deux critères au moins: être rapidement opérationnel et permettre une extrapolation rapide vers un système diffusable.

Le premier critère implique que la programmation d'EDME doit utiliser au maximum les possibilités des outils logiciels, principalement les méta-outils (outils paramétrables ou générateurs), et celles des matériels de façon à diminuer l'effort d'implantation. Cette démarche n'est pas sans soulever quelques questions importantes:

«Quel est le degré de généralité du logiciel obtenu ?»

«Quelles sont sa transportabilité (adaptation à un autre poste de travail) et sa transposabilité (adaptation à un domaine proche) ?»

«Quelle est la dépendance vis à vis des outils employés ?»

Il n'est pas toujours possible de répondre exactement, mais cela ne nous semble pas important. L'essentiel est que ces questions apparaissent constamment en filigrane au cours du développement de la maquette.


```
* fonctions TEXTE --> ARBRE

   - éditeur du texte
   - analyseur syntaxique


* fonctions ARBRE --> ARBRE

   - gestion des déplacements
   - greffe/coupe de sous arbre
   - CONTROLES
   - syntaxique
   - sémantique statique
   - vérificateur de type
   - vérificateur de "bon emploi" des
        récurrences
   - méthodologie
   - vérificateur d'ordre d'entrée des
        définitions
   - autorisation d'emploi d'un objet
   - génération de sous-arbres
        (types, commentaires)
   - gestion d'informations diverses
   - objets restant à définir
   - objets récurrents à initialiser
   - objets externes utilisés dans un
        module

* fonctions ARBRE --> TEXTE

   - INTERFACE UTILISATEUR
        - "décompilateurs"
   - AFFICHAGE
        - "formatage" du texte
        - gestionnaire d'écran
   - IMPRESSION
        - gestionnaire d'imprimante
   - INTERFACE TRADUCTEUR
        - génération de la table des
             symboles
        - génération de la structure
             d'entrée
```

Figure 5. Carte des domaines d'EDME

Le second critère impose de bien clarifier et séparer les différentes fonctionnalités et les *domaines* de leur application. La figure 5 présente la *carte des domaines* d'EDME.

Cette classification permet de traiter deux aspects essentiels d'une maquette:

l'évolution. Une fonction de l'éditeur s'analyse par les domaines qu'elle recouvre et un assemblage des primitives propres à chacun.

la spécification. Spécifier revient à définir formellement les primitives de chaque domaine puis, pour chaque fonction, leur assemblage.

3.2.2 La structuration

Un texte, lorsqu'il est manipulé à l'aide d'un éditeur, acquiert une forte structure. Nous connaissons quatre types de structures:

La ligne. C'est la structure des premiers éditeurs, représentation immédiate des paquets de cartes perforées.

Les entités lexicales. Les éditeurs *video* (citons EMACS, WINNIE, TED par exemple) permettent souvent de manipuler des entités déterminées par des critères lexicaux. Il est ainsi possible de se déplacer d'un caractère (l'entité la plus simple), d'un mot, d'une phrase ou d'un paragraphe. Eventuellement, certaines entités spécifiques sont reconnues (les commentaires par exemples) lorsque l'éditeur est adapté à un langage donné. Le critère de définition est simple: une entité est comprise entre des séparateurs spécifiques; par exemple, un mot est ce qui se trouve entre deux caractères non alphanumériques, une phrase, ce qui est encadré par deux caractères de ponctuation, un paragraphe, ce qui est entre deux lignes blanches, ...

L'arbre abstrait. Les entités manipulées sont définies ici par une grammaire. On éditera donc une instruction, une expression, une procédure, un bloc par exemple. La représentation "naturelle" de telles entités est l'arbre syntaxique.

La structure propre à un langage. Nous pensons ici aux éditeurs intégrés aux langages LISP ou SMALLTALK par exemple). Les entités correspondent alors à des concepts du langage (liste, paire pointée ou objet, méthode, message, par exemple).

Il faut bien noter ici que cette structure est celle perçue par l'utilisateur. La perception se fait essentiellement par les commandes mises à disposition du programmeur. De tous les éditeurs que nous avons utilisés (ED (UNIX),QEDX (MULTICS), EMACS, WINNIE, MENTOR, éditeur bVlisp) il apparaît que la structure externe du texte est très proche de sa structure interne (données effectivement manipulées par l'éditeur). Ceci n'est pas sans conséquences; prenons par exemple l'édition d'un programme avec MENTOR: l'utilisateur est obligé d'avoir trois représentations différentes de son programme: l'arbre abstrait, sur lequel sont définies les commandes, le texte, tel qu'il apparaît sur l'écran et enfin la structure *fonctionnelle* de ce que doit effectuer le programme. La difficulté est ici de maintenir une cohérence entre ces trois vues.

Avec EDME, nous désirons que l'utilisateur n'ait à manipuler qu'une structure unique, charge à l'éditeur de maintenir les liens entre la *vue* de l'utilisateur et la structure interne éditée. Nous pensons que cette séparation des stuctures est indispensable pour rendre EDME accessible à des non-spécialistes.

La structure choisie est calquée sur celle des algorithmes. Nous définissons un algorithme par une hiérarchie d'ensembles emboités: les **couches**. Une **couche** est définie par un ensemble d'objets visibles et manipulables et, de façon duale, par l'ensemble de commandes qui y ont un sens.

Il y a quatre couches:

Couche Algorithme. Les modules sont manipulés dans leur ensemble. Les commandes disponibles sont principalement des demandes d'information comme l'arbre hiérarchique des modules.

Couche Module. Les définitions sont alors visibles et il est possible de les créer, de les supprimer, de les visualiser, ...

Couche Définition. Les constituants internes (mots clés, identificateurs) d'une définition donnée sont alors accessibles.

Couche Environnement. Ce sont ici des algorithmes qui sont manipulés. Cette couche est surtout utile pour gérer plusieurs algorithmes et éventuellement permettre leur réemploi total ou partiel.

Cette structuration simple est cependant suffisante pour se déplacer dans un algorithme, le créer et le modifier. Elle offre en outre l'avantage de ne confronter l'utilisateur qu'à un seul concept à la fois.

Afin de rester cohérent avec le principe de l'unicité de structure, la couche Définition est composée uniquement d'un éditeur de texte, toujours activé depuis la couche Module. Ainsi, quoiqu'EDME soit bâti sur un éditeur syntaxique, l'utilisateur ne manipule et ne voit que du texte.

Chapitre 4

La réalisation d'EDME

Dans ce chapitre nous présentons l'éditeur d'algorithmes, EDME, tel qu'il peut être actuellement utilisé. Comme nous l'avons déjà dit, EDME est une maquette, c'est à dire que la version actuelle est amenée à subir de nombreuses modifications, tant en ce qui concerne les commandes (leur nombre, leur spécification, leur structuration, ...) qu'en ce qui concerne l'interface utilisateur (nature du poste de travail, outils disponibles, ...). Afin de faciliter les modifications ultérieures, il est essentiel que les choix de réalisation soient explicités et critiqués. Nous avons donc choisi de présenter EDME selon cette optique, une description critique des choix, plutôt que selon l'optique d'un manuel d'utilisation, plus adaptée à un logiciel opérationnel.

La première section décrit les commandes et leur structuration telles qu'elles sont visibles par l'utilisateur. Ce dernier point est bien évidemment inspiré par les analyses décrites aux chapitres 2 et 3.

La seconde section propose une *grille d'analyse* et des critères utiles pour choisir un logiciel de développement. Cette grille résulte de l'expérience acquise lors de la construction d'EDME.

La troisième section présente les commandes sous un angle plus technique. Nous y décrivons en particulier la méthode de lancement retenue, leur structure interne, les fonctions internes qui leur servent de support et les choix généraux d'implantation.

La quatrième section décrit les interfaces utilisateurs : le poste de travail retenu et les outils associés à chaque composante du poste.

4.1 Les commandes d'EDME

Nous présentons ici les commandes de la première maquette d'EDME, actuellement disponible. Rappelons que cette maquette est utilisée comme support d'une étude sur les stratégies de programmation employées par des programmeurs professionnels, menées par J.M. Hoc (Laboratoire de Psychologie de Travail de l'EPHE).

4.1.1 Caractérisation des commandes

Les commandes d'EDME sont conçues pour fournir une réelle assistance à l'utilisateur et pour faciliter l'accès à l'éditeur par les débutants. Les caractéristiques sont donc :

- l'intégration de différentes fonctions (contrôles en particulier)
- la simplicité d'emploi
- la cohérence avec la structure éditée

Les deux premières caractéristiques sont essentiellement traduites par le fait qu'une commande d'EDME correspond à une action *logique* sur un algorithme (création d'une algorithme, adjonction d'une définition, ...) et qu'elle est lancée par un nom unique. Les paramètres éventuellement nécessaires (désignation d'une définition par exemple) sont demandés à l'utilisateur via un court dialogue; les actions impliquées (adjonction, suppression de texte, contrôles divers, demande de documentation, par exemple) sont automatiquement activées par la commande.

Nous avons choisi de présenter les algorithmes MEDEE selon une structure simple *d'ensembles emboités* : les couches (cf Chapitre 3 section 2.2). Afin de maintenir une cohérence entre la structure de l'éditeur et celle de l'algorithme, les commandes sont organisées elles-aussi en couches. Sur une couche, seules les commandes manipulant les entités *visibles* sont accessibles. Il faut toutefois noter que certaines fonctionnalités sont communes à plusieurs couches comme, par exemple, le passage d'une couche à une autre ou le nettoyage de l'écran sali par des parasites. Deux solutions sont alors envisageables : soit différencier ces commandes selon la couche, ce qui accroît notablement le nombre de commandes, soit avoir une catégorie de commandes accessibles en permanence, ce qui peut induire une certaine *déstructuration* de l'éditeur. Nous avons choisi la seconde solution, estimant qu'un faible nombre de commandes est un facteur positif lors de la phase d'apprentissage de l'outil.

Nous avons donc deux catégories de commandes :

- celles qui sont structurées en couches (commandes **structurées**)
- celles qui sont toujours accessibles (commandes **transversales**)

Le classement d'une commande dans l'une ou l'autre des catégories est un choix de conception qui n'est pas toujours simple. Les commandes et leur classement que nous présentons ci-après correspondent à la maquette utilisée lors des expériences psychologiques.

4.1.2 La couche Environnement

Actuellement peu développée, cette couche permet d'archiver, de restaurer et d'initialiser la création des algorithmes. Les commandes actuelles sont :

.CRALG
Pour commencer la création d'un algorithme Tout algorithme présent en mémoire est irrémédiablement perdu si on n'a pas pris soin de le GARDER. On entre automatiquement dans le module principal.

.RESTO
Permet de relire un algorithme sauvegardé lors d'une session précédente.

.QUITTER
Pour sortir de EDME. Ne fait pas de sauvegarde systématique de l'algorithme courant.

.GARDER
Pour sauvegarder l'algorithme courant. Si un algorithme de même nom existait déjà en bibliothèque, il est "écrasé" par le nouveau.

.IMPRIME
Imprime l'algorithme courant (s'il existe) sur l'imprimante.

Dans les versions futures d'EDME, cette couche devra être augmentée de fonctions permettant de lister les algorithmes déjà construits et éventuellement de fonctions permettant de manipuler plusieurs algorithmes simultanément pour en extraire des parties réutilisables dans un algorithme en cours de construction.

4.1.3 La couche Algorithme

En MEDEE, les modules sont créés et manipulés avec les définitions qui introduisent leur nom, de ce fait la couche Algorithme est elle aussi peu développée. Les commandes permettront l'observation de l'état des modules et de leur relation (impression des modules à terminer et de l'arbre des modules).

4.1.4 La couche Module

L'essentiel du travail de création d'un algorithme se déroule à ce niveau (adjonction, création, suppression de définitions) et cette couche comporte donc les commandes les plus importantes d'EDME.

Les commandes actuelles sont :

.CRDEF
Pour entrer une définition formelle. Le système se charge de questionner l'utilisateur pour construire les définitions informelles associées.

.INSERER
Réalise une fusion de définition. Ajoute un objet à définir en partie gauche d'une définition existante.

.SUP
Pour supprimer une définition formelle d'un objet. Si l'objet apparaît dans une liste en partie gauche de la définition, il est simplement retiré; sinon, la définition est supprimée et avec elle, éventuellement, une descendance de modules.

.SUPLEX
Pour supprimer la définition lexicale d'un objet. Ce n'est autorisé que si l'objet n'est plus employé NULLE PART dans l'algorithme.

.REMONTER
Sert à remonter un objet introduit dans un module mais qui est une constante pour ce module et doit être défini dans son module père, voire plus haut dans l'ascendance. Cette remontée n'est possible que si l'objet est seulement défini lexicalement (restriction temporaire).

Actuellement, une modification directe textuelle ou arborescente d'une définition n'est pas possible. Ceci est volontaire de façon à forcer l'incrémentalité de l'édition : une modification est une destruction suivie d'une création.

4.1.5 La couche Définition

Cette couche n'est composée que d'un éditeur de texte et n'est accessible qu'à travers des commandes de la couche Module. Il s'agit là encore d'un choix lié à l'incrémentalité.

Nous renvoyons le lecteur à la dernière section de ce chapitre pour le listage des commandes de l'éditeur de texte.

4.1.6 Les commandes transversales

Ces commandes permettent de se déplacer entre les couches et dans l'algorithme. Elles permettent aussi de "rattraper" les situations confuses (écran sale, système ou utilisateur perdu). La commande **.MAISON** nous semble importante dans un système comme EDME. En effet, l'éditeur entretient des liens entre la structure externe (visible par l'utilisateur) et la structure interne ainsi que différentes informations. Il peut arriver que certains de ces liens soient détruits (par exemple, par une activation directe malencontreuse de MENTOR) ou modifiés. EDME contrôle donc à chaque commande que ses pointeurs sur l'arbre abstrait correspondent bien à la couche actuelle; en cas d'échec de la vérification, il se déclare *perdu*. La commande **.MAISON** permet de restaurer les liens et de réinitialiser l'édition.

Les commandes actuellement disponibles sont :

.ALLER

Pour aller directement dans un module. Suppose qu'un algorithme existe.

.MONTER

Pour passer sur la couche supérieure. Echoue si on se trouve sur la couche Environnement.

.DESC

Pour passer sur la couche inférieure. Echoue si on se trouve sur le niveau Module.

.NOMTYP

Pour nommer après-coup le type structuré d'un objet ou d'un champ.

.MAISON

Lorsque le système est perdu (il vous l'aura indiqué !) ou que vous êtes perdu, cette commande ramène au niveau Environnement SANS PERDRE l'algorithme édité.

.REAFFICHE

Lorsque l'écran est très sale (parasite, erreur dans les touches de *scrolling*, ...), permet de le rafraîchir. A noter que les messages d'erreurs rémanents sont perdus ainsi qu'une éventuelle définition dans la zone "temporaire".

.DEMO

Explique les significations des différentes fenêtres.

.RAFFRAICHIR

Efface les messages rémanents

.REVIENS

Commande à activer lorsqu'après une déconnexion accidentelle il a été possible de se remettre sous le système (restaure les paramètres de la ligne de communication)

4.2 Le choix d'un logiciel de développement

La production de logiciel est de plus en plus dépendante d'autres logiciels. De fait, on est amené à choisir un logiciel de développement (MENTOR, CEYX, Gandalf, ...) plutôt qu'un langage de programmation. Depuis deux ans nous avons participé à des degrés divers à plusieurs projets dans lesquels un éditeur

syntaxique tient une place importante : citons **Maiday**, **SPES** (FIN,84b) (BUY83), **LASSIF** (LEC,84). Les deux outils employés sont META−MENTOR et CEYX. Nous renvoyons le lecteur à (KAH,83) et (MEL,82) pour une description de META−MENTOR et à (HUL83a) (HUL83b) pour CEYX. L'objet de cette section est de présenter les critères de choix d'un éditeur syntaxique qui nous semblent pertinents.

Dans cette section, nous appelons *système* le logiciel qui est développé en utilisant les *outils* disponibles par ailleurs. Un *générateur* est un *méta-outil* qui construit un outil à partir d'une description statique fournie par l'utilisateur.

4.2.1 Pourquoi utiliser un éditeur syntaxique ?

Cette question est bien évidemment la première que l'on doive se poser. En effet un éditeur syntaxique est un outil complexe, lourd à mettre en oeuvre (par rapport à un éditeur de texte) et coûteux, tant à l'exécution qu'à l'apprentissage. Deux raisons justifient ce choix :

1. Au cours de l'édition, des traitements importants sur le "texte" sont réalisés (contrôles sémantiques, transformations guidées par l'utilisateur par exemple).
2. Le formalisme implanté est nouveau et doit être mis au point.

Le point (1) tient au fait que la structure éditée, un arbre abstrait, est la plus adéquate pour manipuler un texte formel. En dispensant d'une phase d'analyse syntaxique explicite, en facilitant l'accès aux structures du langage, en simplifiant les déplacements à travers le texte et en permettant la génération de morceaux de texte, un éditeur syntaxique simplifie la programmation des fonctions et permet une réalisation rapide.

Le point (2) est très important dans le domaine de la recherche où la définition de formalismes est une activité constante. Pour être "bon", un formalisme doit être simple d'emploi, aisément implantable et pertinent vis-à-vis de son domaine d'application. Pour ce qui concerne cette dernière qualité, un éditeur syntaxique n'est d'aucun secours, par contre en simplifiant le maquettage d'un système, il accélère l'étude d'implémentabilité. La première qualité mérite un arrêt. Un formalisme nouveau correspond souvent à l'émergence de nouveaux concepts et nous considérons qu'un formalisme est simple (agréable à employer et avec des structures bien adaptées) lorsque les concepts sous-jacents sont clairs. L'intérêt d'utiliser un éditeur syntaxique est alors double : en obligeant le concepteur à dessiner une grammaire, il permet de fixer les idées, ensuite, en permettant à des utilisateurs de manipuler très rapidement des textes au niveau des structures du langage, il permet la découverte des lourdeurs, des zones obscures et, le cas échéant, des incompatibilités. Le cycle de construction d'un formalisme, formé des trois étapes : formulation des concepts, traduction en des structures syntaxiques et validation sur des exemples, est alors raccourci et devient un processus quasiment interactif.

4.2.2 Critères syntaxiques

L'utilisation d'un générateur d'éditeur syntaxique nécessite la description des entités de base qui sont, dans la terminologie de MENTOR, les **opérateurs** (i.e. les noeuds de l'arbre abstrait) définis par une arité et un nom, et les **phyla**. Un phylum est l'ensemble des opérateurs greffables en un certain point de l'arbre. La terminologie de CEYX emploie respectivement les **constructeurs** et les **univers**. L'aspect intéressant est ici la lisibilité de cette description. En séparant clairement la définition des opérateurs et des phyla, MENTOR, à travers le langage METAL, nous semble sur ce point préférable à CEYX dans lequel nous ne connaissons les éléments d'un phylum que par le parcours des définitions des opérateurs le composant.

Un second point concerne les relations entre la syntaxe abstraite et la syntaxe concrète. Dans le sens concret $--> $ abstrait (l'analyse syntaxique) l'existence de générateurs d'analyseurs syntaxiques permet une définition conjointe des deux grammaires. Dans le sens abstrait $--> $ concret, une description statique de la grammaire augmentée de directives de formatage peut être utilisée si un *méta-décompilateur* adéquat est disponible, sinon, il faut programmer directement le décompilateur. Là encore, METAL est supérieur à CEYX en intégrant les deux descriptions dans un même cadre. Cet aspect est important tant pour le réalisateur parce qu'il offre une vue globale du langage, une référence et facilite les modifications en liant les divers constituants, que pour l'utilisateur final en fournissant une documentation exacte et lisible sur les grammaires et leurs relations.

Dans le cadre du projet **LASSIF**, réalisé à l'aide de CEYX, nous avons conseillé l'utilisation de METAL pour avoir une première définition du formalisme qui a ensuite été traduite sans problème en une description CEYX. Cette démarche s'est révélée intéressante par la séparation entre les problèmes de définition et d'implantation du langage et par l'utilisation des qualités de chaque outil : clarté de METAL, puissance de CEYX dans la mise au point.

4.2.3 Critères sémantiques

Nous avons vu que l'édition des programmes intègre de plus en plus de contrôles, les vérifications de types ou les doubles déclarations par exemple pour EDME. Il serait donc intéressant d'avoir des générateurs de vérificateurs utilisant une description statique des contraintes contextuelles. Malheureusement de tels outils ne sont pas encore opérationnels à notre connaissance. Des formalismes utilisant des règles d'inférences comme TYPOL (DES,83) ou des grammaires attribuées (DEM,81) sont actuellement expérimentés. Nous devons donc nous contenter de descriptions sous formes de *fonctions sémantiques* (au sens des analyseurs syntaxiques) de ces contraintes. Il convient d'être très vigilant et de ne jamais perdre de vue qu'à partir de l'instant où des programmes ont été écrits, la modification de la grammaire devient très délicate. En effet,

le code développé travaille sur une certaine structure de l'arbre abstrait que tout changement va modifier, invalidant de ce fait beaucoup de programmes.

Le critère de choix va donc ici porter sur la puissance du langage de programmation propre à l'éditeur et son adéquation aux traitements prévus : certaines applications privilégient les parcours d'arbre qui devront alors être simples et performants, d'autres utilisent des structures de données complexes (tables de symboles, graphes de dépendance) pour lesquelles le langage de programmation doit être bien adapté.

4.2.4 Critères d'extensions à d'autres systèmes

En général, l'éditeur syntaxique n'est qu'un maillon d'une chaîne dans le système final et il faut alors connecter les composants entre eux. Selon la situation d'un composant dans la chaîne, les besoins de connexion seront distincts : un composant en amont de l'éditeur devra pouvoir construire un arbre abstrait, en aval, il devra surtout consulter et parcourir la structure. Enfin, certains devront interagir avec l'éditeur en activant directement certaines commandes et en échangeant des informations.

Nous classons les éditeurs syntaxiques en deux groupes :

- les systèmes clos
- les systèmes intégrés à un langage

Un système clos est un système spécialisé dans l'édition qui ne donne pas accès à ses fonctions internes. MENTOR est un système clos. Les communications avec l'extérieur ne peuvent s'effectuer que via un interface restreint. L'avantage principal d'une telle conception réside dans la sécurité : l'éditeur contrôle toujours la cohérence de son état et de la structure éditée. L'inconvénient tient à la difficulté de programmer (ou de réutiliser) des composants externes interagissant avec l'éditeur.

Un système intégré est un ensemble de primitives accessibles depuis un langage donné, c'est un *sur-système* d'un environnement de programmation existant. CEYX, en utilisant LISP, et GANDALF avec C, sont des éditeurs intégrés. La programmation des autres composants est alors simplifiée : l'accès à la structure et aux fonctions est immédiat, l'environnement offre des aides à la programmation supérieures à celles d'un langage propre à un système clos et il est possible de récupérer des logiciels déjà existants. Le danger potentiel inhérent à cette conception est le risque de rendre incohérent l'état de l'éditeur ou la structure.

4.2.5 Les critères d'interaction avec l'utilisateur

Nous voyons principalement trois domaines :

1. l'entrée du programme
2. l'entrée des commandes
3. l'affichage du texte

Concernant le point (1), deux grandes options sont possibles : entrée arborescente ou entrée *textuelle* suivie d'une analyse syntaxique. L'entrée arborescente semble naturelle lorsqu'on emploie un éditeur syntaxique mais elle peut devenir très lourde lorsqu'on définit les expressions élémentaires (2+**alpha*gamma** par exemple) et ne se justifie à notre avis que dans des cas particuliers, par exemple si les règles de priorité des opérateurs sont très complexes et inhabituelles. Il faut alors considérer ici la facilité de connecter un analyseur syntaxique. La figure 6 présente les deux types d'entrée.

Le point (2) est relativement important car un éditeur syntaxique peut avoir un nombre élevé de commandes, surtout si à chaque schéma syntaxique est associée une commande. Dans MENTOR, les schémas syntaxiques sont nettement séparés des commandes qui sont alors peu nombreuses. Ceci implique que les lignes de commandes que doit frapper l'utilisateur sont longues (commande *plus* paramètres). A l'inverse Gandalf, le CPS ou CEYX cherchent à minimiser le nombre de caractères frappés. L'approche de CEYX nous semble très intéressante en reprenant la notion de *clés* d'EMACS (à chaque caractère frappé au clavier est associée une fonction) qui permet à l'utilisateur de se construire son propre ensemble de clés. On est alors confronté au problème de la multiplication des commandes et il faut prévoir des systèmes d'aide pour cet aspect.

Le point (3) concerne les fonctions que fournit le générateur pour la décompilation et l'affichage de l'arbre. Il faut regarder ici la puissance et la richesse des fonctions proposées. Des fonctions puissantes simplifieront la programmation du décompilateur mais risqueront de contraindre la forme externe du langage; ainsi, pour EDME, nous n'avons pas pu utiliser les fonctions de MENTOR, ces dernières étant inspirées par un langage "linéaire" comme PASCAL, alors que MEDEE nécessite trois colonnes indépendantes. A l'inverse, un interface plus pauvre augmentera le travail de programmation mais offrira plus de souplesse.

l'utilisateur frappe	le sytème effectue	l'écran affiche
	entrée textuelle	
.Entre X: si A=B alors MOD1 sinon MOD2	l'édition du texte l'analyse syntaxique la greffe d'arbre la décompilation	X : si A = B alors MOD1 sinon MOD2
	entrée arborescente	
.structure	vérifie si la commande est licite greffe le schéma va sur le premier fils	‹t_id› : <struct>
.ident(X)	id. passe au second fils	X : ‹struct›
.si	id	X : si ‹cond› alors <Act> sinon <Act>
.=	id	X : ‹exp› = <exp> alors <Act> sinon <Act>
.ident(A)	id	X : A = ‹exp› alors <Act> sinon <Act>
	etc	

La position du curseur structurel est indiquée en fonte *Venice*.

Figure 6. Entrée arborescente vs textuelle

4.2.6 Les critères divers

Sous cette rubrique, nous nous contenterons de citer quelques autres critères qui sont très dépendants de l'application finale souhaitée.

Pour certains types d'applications, la rapidité de réponse peut être une qualité importante. Il faut alors considérer les techniques de réalisation et les langages support des générateurs; il est prévisible que MENTOR écrit en PASCAL (donc compilé) sera plus rapide que CEYX qui utilise l'interprète LISP.

D'autres applications nécessitent de manipuler plusieurs langages en même temps. Les deux générateurs que nous utilisons le permettent mais avec des contraintes diverses : avec MENTOR, il n'est actuellement pas possible d'avoir plusieurs analyseurs syntaxiques dans la même session.

Enfin, les annotations, commentaires ou attributs peuvent être utilisés plus ou moins facilement suivant les éditeurs et les applications. Là encore une analyse fine et spécifique de chaque générateur doit être effectuée.

4.3 L'implantation des commandes

4.3.1 Le lancement

Comme tout système interactif, EDME utilise un interprète de commandes. Afin d'aboutir le plus rapidement possible à une maquette utilisable, nous avons choisi d'employer directement l'interprète de MENTOR dans la première version. Parmi les conséquences de ce choix, citons :

Une diminution de l'effort de programmation et la possibilité de tester très rapidement les nouvelles fonctions. Cet avantage a été le critère déterminant du choix.

L'obligation de respecter la forme des commandes MENTOR (préfixage par un point, pas d'emploi de *clés*, ...)

Le danger que l'utilisateur active une fonction normalement cachée, détruisant ainsi la cohérence d'EDME.

La difficulté de gérer des informations *rémanentes* d'une commande à l'autre (rémanence des messages d'erreur, informations sur l'utilisateur comme son niveau d'expertise, ses erreurs fréquentes, ...) du fait de la distribution du code à l'intérieur de chaque commande.

Il est clair qu'à terme, un interprète de commandes spécifique à EDME devra être implanté.

Les commandes sont lancées via un nom unique, les différents arguments nécessaires, comme la désignation d'une définition pour une suppression par exemple, sont demandés à l'utilisateur par un petit dialogue. Ceci reste cohérent avec la conception d'EDME comme un *guide*.

4.3.2 La structure générale d'une commande

Les commandes structurées sont construites sur le schéma de la figure 7

Le point essentiel à noter ici est l'absence totale de modification de l'algorithme tant que tous les contrôles n'ont pas été couronnés de succès. En corollaire, un échec à l'un quelconque des contrôles fait perdre tout le travail effectué jusque-là dans la commande, en particulier le texte entré par l'utilisateur. Ce point peut être gênant dans le cadre de l'implantation actuelle car nous ne disposons pas encore de mécanisme permettant de récupérer ce travail et la perte totale peut être très frustrante pour l'utilisateur.

Chaque fonction gère elle-même les messages d'erreur, de demande d'information et l'entrée des données lorsqu'ils lui sont propres.

L'activation des traitements propres n'est pas ici détaillée car elle est spécifique à chaque commande. Les fonctions de base (recherche dans l'arbre, contrôles de types, affichage, ...) sont indépendantes des commandes. Chaque commande gère et contrôle l'appel à ces fonctions selon les besoins spécifiques. Le contrôle est généralement une analyse par cas portant sur la situation à traiter (type de définition, type de module, couche actuelle, ...).

La figure 8 présente une commande (relativement simple) d'EDME accompagnée de quelques explications.

```
DEBUT
Analyse d'autorisation (couche, cohérence de l'arbre)
Restauration de l'affichage

SI Succés
 ALORS

 Entrée des paramètres % éventuellement %

  SI Succés
   ALORS

   Activation des traitements propres
   Construction de sous-arbres intermédiaires
       (si nécessaire)

  FINSI
 FINSI

 SI Succés
  ALORS

  Intégration des sous-arbres à l'algorithme mémorisé
   (si nécessaire)
  Destruction des sous-arbres
   (si nécessaire)
  Mise à jour des informations non visualisées

  SINON

  Emission des messages d'erreur
 FINSI

 Restauration de l'affichage

FIN
```

Figure 7. Le schéma d'une commande


```
% ==========================================================================

% --------------------------------------------------------------------------
%         Remontée d'un objet dans un module englobant
%         Cet objet doit être défini lexicalement dans le module courant
%         mais non défini formellement
% --------------------------------------------------------------------------
  ( ( .SETFLAG ; .EFFACE<@ENTREE>; %== restauration affichage

     ( @idcour = @l-ident ; @idcour s ;

          %== lecture de la définition a remonter

% On recherche l'objet dans le lexique du module courant
           %== debut du traitement propre

       @l = @lex-ident ; @l s c @idcour ;
       @l1 : @modcrt ; @l1 f @l ; ? ,
         ( .mes2<@objinexist> ; $4 ) ; @l : @premdef ;
         ( @l2 : @l s2ss ;
           ( .eqval<@idcour,@l2> ; ? $ ; @l2 r )* ; ?
             ( .mes2<@nonrem> ; $4 ) ; @l r )* ; ? ;

   %== Les contrôles sont validés
   %== La définition est insérée dans l'arbre

            @modup = @l-ident ; @modup s ; .mes2<@quelmod> ; .LIT<@modup> ;
            ? , $-3 ; @l : @modcrt s ;
          ( .pere<@l,@l2> ; .eqval<@modup,@l2 s> ; ? $ , @l:@l2 s )* ; ?
          ( @l2 s2s ; ( .eqtype<@lex-ident,@l2> ; @l2 r )* ; ? ;
          @l2 j @l1 ; .supadef<@idcour> ; @modcrt pd ;
          @modcrt : @l2 u2 ;.putadef<@l1> ; @modcrt pu ;
          d @l1  ; @kk : @modcrt s2s ; ? , @modcrt s2 c@l-def-lex ),

    %== Remise en ordre de l'affichage et émission message d'erreur général
    %==  (Si nécessaire)

;
```

Figure 8. Commande commentée

4.3.3 Les différentes classes de fonctions

Nous avons cherché à séparer dans l'implantation les commandes et les fonctions, ceci pour faciliter l'évolution des commandes et la spécification d'EDME. Chaque classe de fonctions traite d'un domaine particulier d'une façon indépendante des autres classes.

Nous avons actuellement cinq classes principales :

1. La gestion de l'affichage et de l'entrée de l'utilisateur. Elle permet de spécifier la fenêtre de destination, les constituants du messages, les effacements, ...
2. Les recherches dans l'arbre. Elles sont assez synthétiques comme par exemple rechercher le module *père* d'un autre, le type d'un objet donné, ...

3. Le contrôleur de type.
4. La gestion des *avec*. Cette information est utilisée tant à titre documentaire pour l'utilisateur que pour gérer les suppressions d'objets et de définitions.
5. La gestion des initialisations. Ceci est une aide appréciable pour l'utilisateur qui n'a plus à se soucier de déterminer les objets à initialiser lors des itérations.

   Ces classes reflètent l'organisation actuelle qui devra être améliorée par :

6. Une classe reprenant les contrôles sémantiques et méthodologiques actuellement trop dispersés dans le code.
7. La gestion des réutilisations d'arbre. En effet, lorsqu'une commande avorte, on perd toutes les informations rentrées (définitions, types, commentaires, ...) ou lorsqu'on détruit une définition, on peut perdre toute une hiérarchie de modules. Le ré-emploi est complexe car nous souhaitons toujours conserver l'incrémentalité d'EDME en garantissant la *correction* de l'algorithme.
8. Une (ou plusieurs) classe(s) gérant l'inférence d'informations (type des objets en particulier).

Nous verrons plus loin, au chapitre 7, comment nous voyons l'évolution d'EDME et les nombreuses nouvelles classes qu'elles va amener à définir.

4.3.4 Les techniques employées

Nous ne ferons pas ici le détail des techniques employées pour la programmation des différentes fonctions d'EDME mais nous présentons les choix importants qui ont guidé la réalisation.

Notre premier choix a été de rester entièrement dans le cadre de MENTOR : en particulier la programmation est réalisée totalement en MENTOL, le langage de programmation interne à MENTOR. Les avantages sont multiples :

- pas de problème d'interface MENTOR–autre outil
- excellente adéquation des instructions MENTOL aux traitements d'arbres
- pas de problèmes de mise à jour en cas d'évolution de MENTOR (tant que les spécifications externes ne changent pas évidemment)
- transport sur une machine possédant MENTOR quasi immédiat.

Il existe évidemment quelques inconvénients, citons :

- la faiblesse des outils de mise au point

- le caractère *cryptique* du code MENTOL
- une rapidité d'exécution moyenne.

Le second choix a été de réduire au maximum l'information rémanente autre que l'arbre abstrait. Les éléments rémanents sont conservés dans des *attributs* de l'arbre abstrait, réalisés à l'aide du mécanisme d'annotation de MENTOR. Par conséquent, il est nécessaire de reconstruire beaucoup d'information lors de chaque commande. La raison principale de ce choix réside dans la difficulté qu'il y a à maintenir la cohésion entre des données dispersées tant au cours d'une même session qu'au cours des sauvegardes et restaurations d'algorithmes. L'inconvénient majeur est bien sûr la lenteur du traitement.

Nous estimons que ce dernier point n'est en fait pas important en ce qui concerne EDME pour deux raisons :

1. l'utilisateur type d'EDME est un débutant, donc ses algorithmes seront courts; les parcours d'arbres sont alors brefs et les temps de réponses très corrects. Nous avons noté que, sur Multics, EDME passait plus de temps à faire les analyses syntaxiques qu'à exécuter le code MENTOL avec des algorithmes de taille moyenne.
2. EDME doit être employé pour créer des algorithmes sans qu'auparavant ceux-ci n'aient été écrits sur papier. Il est donc prévisible que lors d'une session, le temps de réflexion de l'utilisateur prédomine largement sur le temps de calcul de l'éditeur. Nous pensons même, mais cette hypothèse reste à vérifier, qu'un temps de réponse trop rapide peut nuire : un système qui "pousse" l'utilisateur ne lui laisse pas le temps de bien réfléchir; évidemment, les temps ne doivent pas non plus devenir prohibitifs sous peine de lasser le programmeur.

Le dernier principe que nous appliquons est d'utiliser le plus possible les fonctionnalités de MENTOR, en particulier, les mécanismes d'annotation qui permettent de maintenir de l'information très structurée dans l'arbre tout en la cachant à l'utilisateur. Cette utilisation des annotations est conjointe à une réflexion de l'équipe sur l'utilisation de grammaires attribuées pour spécifier et réaliser EDME.

4.4 Les interfaces avec l'utilisateur

4.4.1 Le poste de travail

La définition du poste de travail a tenu compte d'impératifs techniques mais aussi *ergonomiques*. Ce dernier point est important dans le cadre d'applications interactives car d'une part le programmeur, novice dans

notre cas, doit avoir une première impression favorable : l'utilisation d'EDME en dépend, d'autre part, les sessions seront longues, ce qui conduit a essayer de ralentir la fatigue du programmeur.

N'ayant que peu de prise sur l'aspect matériel, nous nous sommes contentés de choisir au mieux parmi les périphériques disponibles sur le marché. Le poste Maiday est constitué d'un terminal alphanumérique et d'une imprimante. Les caractéristiques minimales sont :

- pour le terminal :
  - largeur d'écran de 132 caractères (rendu obligatoire pour afficher les trois parties d'un algorithme)
  - bonne lisibilité sur cette largeur
  - possibilité de visualiser un module dans son intégralité
  - facilité de programmer un multi-fenêtrage
  - disponibilité de touches de fonctions programmables
- pour l'imprimante :
  - vitesse d'impression suffisante (supérieure à 100 cps)
  - qualité d'impression
  - possibilité de changement de police de caractères
  - facilité de connexion au terminal (travail sur réseau)

Les périphériques retenus sont un terminal TAB 132/15 qui offre 24 lignes d'affichage, 96 lignes de mémoire consultable localement, gestion de trois *bandes* d'affichage (un rouleau (*scrolling zone*) et 2 bandes fixes) de hauteur programmable. Les caractéristiques de ce terminal que nous utilisons correspondent à une norme ANSI, ce qui autorise l'emploi d'autres terminaux sans aucune modification dès lors qu'il sont conçus selon cette norme. L'imprimante est connectée directement au terminal.

Chacun des trois composants (écran, clavier, imprimante) possède son propre interface.

4.4.2 L'écran

La figure 9 présente l'écran typique tel qu'il apparaît lorsque l'utilisateur crée un algorithme.

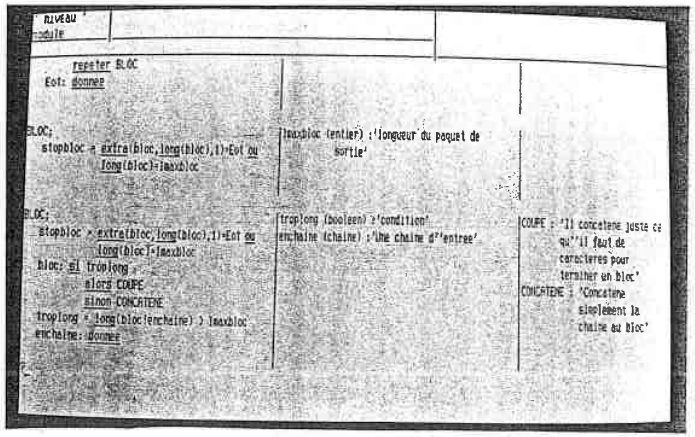


Figure 9. Ecran typique

Techniquement, nous avons fait deux choix fondamentaux :

1. tout texte apparaissant sur l'écran est un arbre décompilé
2. EDME ne gère pas d'image interne de l'écran

Le choix (1) est lié à notre volonté de rester le plus possible à l'intérieur de MENTOR. Il présente le gros avantage de permettre la construction de messages intégrant du texte et des arbres en cours d'édition. Cette possibilité, encore assez peu utilisée, devrait permettre d'envoyer à l'utilisateur des messages d'erreur explicites. L'autre avantage, plus technique, est de limiter le développement à un nombre restreint de composants d'affichage.

Le choix (2) est lié à notre volonté d'utiliser au maximum les outils à notre disposition et de limiter l'effort de programmation. Il est évident que nous perdons sur la généralité et la portabilité de Maiday mais nous constatons que le nombre de terminaux disposant des fonctionnalités que nous employons s'étend sur le marché. Lors de démonstrations, nous avons utilisé EDME sans aucune modification sur différents terminaux répondant aux spécifications de la norme ANSI. Les inconvénients principaux sont de deux ordres : il faut, lors de toute modification, réafficher l'intégralité du module, ce qui ralentit l'interaction, et il peut y avoir perte d'information (principalement en ce qui concerne les messages) lors d'un réaffichage complet de l'écran.

Une description technique des composants de l'affichage peut être trouvée dans (JAC,82). Rappelons les quelques idées directrices de l'implantation :

séparation entre la mise en page, la gestion logique des fenêtres, la gestion physique des tampons et la gestion des transmissions

possibilité de régler les tailles des fenêtres (ce ne peut pas être fait interactivement)

mise en paramètre des séquences de contrôle des terminaux.

Dans la version actuelle d'EDME, nous avons deux systèmes d'affichage parallèles : un pour la décompilation des algorithmes en trois colonnes, l'autre pour les décompilations dans les autres fenêtres.

4.4.3 Le clavier

Nous avons déjà dit combien le clavier nous semble un instrument d'usage délicat. Il l'est d'autant plus avec EDME du fait de la faible tolérance du système aux erreurs dactylographiques. En effet un mot mal dactylographié peut introduire soit une erreur lors de l'analyse syntaxique ou des contrôles divers, ce qui conduit à recommencer, soit des objets qu'il faudra éliminer de l'algorithme, ce qui est assez lourd actuellement. Il faut donc que l'utilisateur puisse réellement éditer (au sens de l'édition de texte) ses entrées avant de les soumettre à EDME. Nous avons donc développé un petit éditeur *plein écran* connectable à MENTOR.

Les caractéristiques essentielles de cet éditeur sont :

- une petite taille (tampon d'une dizaine de lignes, affichage sur trois lignes)
- orientation *video* avec déplacement de curseur
- mode *insertion automatique* pour les caractères imprimables, l'utilisateur est ainsi assuré de ne jamais perdre du texte en le recouvrant
- commandes en faible nombre associées à des clés
- coupure automatique des lignes avec décalage des lignes suivantes en cas de débordement
- multi-terminal, l'adjonction d'un nouveau terminal étant guidée et interactive.

La figure 10 présente la liste des clés et des fonctions associées. Le nombre de fonctions est assez restreint pour permettre d'activer les plus fréquentes en utilisant les séquences générées par le pavé numérique en mode alterné (*Alternate Keypad Sequences*) qui équipe désormais de nombreux terminaux.

En plus de son usage d'éditeur de texte, cet outil sera employé comme support de menu pour le lancement des commandes. Ce mode de lancement, indispensable au débutant, permettra d'afficher la liste des commandes disponibles sur la couche actuelle, de déplacer le curseur par les mêmes clés que le mode texte et de valider la sélection en quittant l'éditeur.

	Les Contrôles	Les Escapes
A	DebutDeLigne	DebutDeBuffer
B	CaracterePrecedent	MotPrecedent
C		
D	EffaceCaractere	EffaceMot
E	FinDeLigne	FinDeBuffer
F	CaractereSuivant	MotSuivant
G		
H		
I	CoupureDeLigne	
J		
K	EffaceLigne	
L	Reaffiche	
M	CoupureDeLigne	FusionDeLigne
N		LigneSuivante
O		
P		LignePrecedente
Q		Quitter
R		
S		
T	Interverti	
U		
V		
W		
X		
Y	*Yank*-Ligne	
Z		

Figure 10. Commandes de l'éditeur de texte

4.4.4 L'imprimante

L'imprimante peut sembler un périphérique superflu dans un environnement de programmation conçu pour être interactif; nous estimons qu'elle reste indispensable à EDME dans son état actuel. Parmi les raisons qui justifient l'imprimante, certaines sont technologiques : la faible taille d'un écran interdit de visualiser un algorithme complet. D'autres sont techniques : la commande permettant un affichage "rapproché" de deux définitions distantes n'existe pas. Mais surtout, MEDEE, comme la plupart des langages algorithmiques, a été conçu en dehors d'une utilisation interactive. La structure syntaxique du langage est adaptée au travail sur des algorithmes complets, écrits sur papier : ainsi les relations logiques entre les différents modules ne sont réellement perceptibles qu'en ayant l'intégralité du texte sous les yeux. Nous estimons qu'un système entièrement interactif ne peut être construit que sur un langage spécifiquement conçu pour être manipulé par des outils interactifs. Par exemple, le langage de spécification LARCH développé par l'équipe de J. Guttag ne peut être lu qu'à travers un éditeur spécialisé (ZAC,83) ou encore un langage comme SMALLTALK (GOL,80), de part son caractère modulaire et l'intégration des *classes graphiques* au noyau du système, est essentiellement orienté vers une utilisation entièrement interactive. Ce changement d'optique permet de concevoir des structures de *textes* qui ne soient plus linéaires, facilitant d'autant les manipulations automatiques et augmentant la puissance d'expression des formalismes.

Pour réaliser le composant d'impression, nous avons réemployé les composants d'affichages existants (décompilateur et gestions des fenêtres logiques) et programmé la composition des lignes imprimées. La difficulté provient ici de ce qu'une imprimante ne possède pas *l'adressage direct* du curseur ! La relative facilité de cette implantation ainsi que le réemploi aisé de l'existant nous ont convaincu de la validité des choix d'implantation du système d'affichage.

L'algorithme de la figure 1 a été imprimé avec ce système.

Chapitre 5
La validation

5.1 Le problème général

Une étape importante de la démarche expérimentale que nous avons adoptée concerne le recueil des résultats de l'expérience et leur exploitation. Il nous faut donc préciser les modalités de ces deux points.

Dans le cas d'EDME il convient de confronter l'outil réel aux objectifs fixés et d'évaluer sa pertinence vis à vis des utilisateurs.

Rappelons (cf Chapitre 1) que nous avons situé notre étude dans trois domaines :

- la méthodologie de la programmation, ce point concerne les fonctionnalités du système
- l'ergonomie, ce point concerne l'interface utilisateur
- le génie logiciel, ce point concerne les techniques de développement d'EDME.

Une technique très puissante de validation d'un environnement de programmation consiste à l'utiliser pour son propre développement. Ce *boot-strap* a été employé avec succès par MENTOR, bVlisp et CEYX. Malheureusement nous n'avons pu employer cette technique pour deux raisons majeures : l'inadéquation connue de MEDEE pour les algorithmes interactifs et l'absence d'outil d'exécution d'algorithmes MEDEE. Nous sommes donc contraint de séparer la validation des trois domaines.

En ce qui concerne la méthode de programmation et l'interface utilisateur, la validation s'effectue par le biais des expériences conduites par J.M. Hoc. L'utilisation d'EDME par des utilisateurs extérieurs au projet doit aussi permettre de réaliser des *réglages* de l'éditeur, c'est à dire de modifier éventuellement la disposition de l'écran, d'ajouter de nouvelles fonctions, d'affiner les systèmes d'aide, ... La section 3 de ce chapitre présente cette expérimentation et les premiers résultats qu'elle a fournis.

Une seconde forme de validation des fonctionnalités est également prévue par le biais d'une spécification formelle *a posteriori* d'EDME. Des essais ont déjà permis de montrer la faisabilité d'une telle spécification en utilisant des types abstraits et la démarche déductive appliquée à la spécification (FIN,79). Il sera intéressant de réaliser cet important travail à l'aide de l'éditeur de spécifications qui est en cours de développement à Nancy (cf projet SPES (FIN,84b)).

Les techniques de développement sont assez difficiles à valider. En effet, un outil opérationnel n'est qu'une faible preuve de la qualité des techniques. Des critères «subjectifs» comme la lisibilité du code, ou plus

objectifs comme les facilités de transport et de maintenance nous semblent plus importants. Nous avons donc rapidement recherché à transporter EDME sur une autre machine, sans attendre une version figée. L'expérience du premier transport est relatée dans la section suivante.

5.2 Le transport sur SM90

Il ne se construit plus de système qui ne se prétende pas portable, EDME n'échappe pas à la règle. En fait un transport est toujours possible, quitte à tout réécrire. Dans le cas de logiciels d'application, comme EDME, par opposition aux logiciels "de base", la dépendance vis à vis d'un système d'exploitation et d'une machine est toujours difficile à évaluer. Au cours de notre expérience de transport, nous avons cherché à isoler les différentes étapes, les problèmes et leurs solutions relativement au cas particulier qu'est EDME. La description qui suit, assez technique, présente des outils utiles au programmeur de logiciels d'application et ne traite pas du problème général de la portabilité.

5.2.1 Qu'est ce qu'un transport ?

EDME se décompose en trois grands ensembles : le noyau de l'éditeur syntaxique qui est constitué du programme METAL et du programme d'entrée du générateur d'analyseur lexical, les interfaces constituées de plusieurs modules écrit en PASCAL et quelques procédures PL/1, et les fonctions et commandes programmées en MENTOL. La taille des programmes transportés est d'environ :

- 700 lignes de METAL
- 2000 lignes de MENTOL
- 3000 lignes de PASCAL
- 100 lignes de PL/1

Le transport d'EDME se décompose en quatre étapes :

1. la transformation des textes sources des langages communs aux deux machines
2. le transfert des fichiers
3. la réécriture d'une partie du code, en particulier lorsqu'il est écrit dans des langages non communs aux machines

4. la reconstruction du système.

5.2.2 La transformation des textes

Pour EDME ce point concerne exclusivement les modules écrits en PASCAL. On peut noter que le choix initial d'utiliser le langage MENTOL se révèle très avantageux. En effet il est très peu probable que MENTOL (ou METAL) soit modifié lors du transport de MENTOR, donc les programmes MENTOL (ou METAL) peuvent être directement copiés d'une machine à une autre. La normalisation de PASCAL devrait supprimer ce problème, malheureusement les facilités indispensables pour de gros systèmes (compilation séparée, liaison avec fichiers externes, ...) sont spécifiques à chaque compilateur; de plus, certaines implantations autorisent des variations lexicales (parenthésage des commentaires, opérateur de différence, ...) interdites par d'autres. La transformation peut donc se révéler délicate à faire manuellement.

L'outil idéal pour ce type de travail est MENTOR. Nous avons utilisé les fonctions MENTOL mises au point par l'équipe MENTOR lors de son transport. La procédure principale est simple : elle comporte un paramétrage du décompilateur et un parcours du texte qui permet de déplacer les déclarations des variables externes dans deux blocs de déclarations spéciaux. Les avantages dans l'utilisation de MENTOR sont nombreux, outre la rapidité des transformations, le caractère systématique est appréciable ainsi que la vérification automatique de certaines contraintes : par exemple, le type des variables externes doit obligatoirement être nommé en PASCAL SM90 contrairement au PASCAL Multics, la procédure signale les points de conflit.

En plus des transformations purement syntaxiques évoquées ci-dessus, MENTOR peut aussi être utilisé pour des transformations plus profondes, citons par exemple la transformation des zones VALUE disponibles sur certains PASCAL en séquence d'initialisations plus conformes à la norme.

5.2.3 Le transfert des fichiers

Deux voies technologiques sont utilisables pour le transfert : soit les supports magnétiques externes (bandes, disquettes, ...), soit les communications directes (réseaux). Pour EDME, nous avons emprunté la seconde voie.

Techniquement, les fichiers ont transités de Multics INRIA à la SM90 de l'équipe MENTOR via un VAX. Les procédures utilisées (tv qui utilise les processeurs de courrier inter-site, et trp, processeur de communication d'UNIX à UNIX développé par F. Montagnac) contrôlent l'intégrité de la transmission.

Le choix d'utiliser des réseaux lors de transfert peut avoir une influence sur la programmation; en particulier, il faut essayer de rendre aisément décelable une erreur de transmission qui souvent n'affecte qu'un caractère. Les procédures MENTOL d'EDME ne répondent pas à ce critère : la rapidité du transport était donc fortement liée à la sûreté de fonctionnement des réseaux et des procédures de communication.

5.2.4 La réécriture de code

Les difficultés de cette étape sont certes liées à la taille du code à réécrire mais elles proviennent surtout des communications entre les différents langages employés. Les deux points clés sont l'implantation des données partagées et le passage des paramètres lors des appels de procédures.

Lors du transport d'EDME, la traduction des programmes PL/1 en C n'a pas soulevé de difficultés très importantes et a été rapidement achevée. A ceci nous voyons deux raisons :

1. nous avons fortement limité la taille du code PL/1 en ne recourant à ce langage que pour pallier les lacunes graves de PASCAL : les entrées sorties interactives.
2. PL/1 (respectivement C) a été choisi car il correspond au langage *natif* de Multics (UNIX). De ce fait, les implanteurs de PASCAL ont naturellement programmé le compilateur en PL/1 (C), utilisant ses mécanismes de passages de paramètres et ses structures de données. La communication PL/1 PASCAL (C PASCAL) est donc simple et bien documentée.

5.2.5 La reconstruction du système

Dans le cas d'EDME, cette phase comporte deux étapes : la génération de l'éditeur syntaxique et l'édition de lien complète du système.

Les créateurs de METAL ont choisi d'utiliser les générateurs d'analyseurs syntaxiques existants sur les systèmes opératoires cibles, **SYNTAX** sur Multics et **YACC** sur UNIX. Si l'utilisation de **SYNTAX** est absolument transparente pour l'implanteur (le compilateur METAL étant bien adapté), il n'en est pas de même avec **YACC**. L'obligation de travailler le fichier de sortie de la compilation METAL avec un éditeur de texte est assez contraignante et nous n'avons pu réaliser ces modifications rapidement que grâce à l'appui de l'équipe MENTOR. Nous pensons que ce fait est l'un des points faibles de l'implantation actuelle de MENTOR sur SM90 (ce point est en passe d'être nettement amélioré). Par cet exemple nous tenons à montrer que les dépendances entre outils peuvent être sources de difficultés pour le transporteur et que celles-ci doivent être décelées rapidement dans la conception d'un logiciel.

L'édition de lien d'EDME est une opération à laquelle nous n'étions pas habitué. En effet, le système Multics, par son mécanisme d'édition dynamique rend tout à fait transparente cette opération et masque totalement l'implantation des outils utilisés. L'outil le plus utile pour cette phase a été **MAKE** qui effectue toutes les opérations nécessaires au maintien de cohérence d'un système complexe en observant les dates de modification des fichiers à partir d'une description des dépendances entre les modules. Outre l'intérêt pratique évident de tels outils, il faut noter leur intérêt documentaire car ils obligent à écrire une description exacte et actuelle de l'architecture générale d'un système, document indispensable sitôt passée une certaine taille.

5.2.6 Remarques générales

Une des questions relatives à la portabilité concerne le choix de bâtir EDME comme un sur-système de MENTOR plutôt que de construire un outil autonome : ne risquions nous pas d'avoir un logiciel intransportable faute de diffusion de MENTOR ? Nous pensons au contraire avoir choisi la voie qui permet la diffusion la plus rapide possible : il nous suffit d'attendre que MENTOR soit porté pour que très rapidement EDME suive. Si nous devions porter EDME sur une machine ne disposant pas de MENTOR, nous sommes convaincus que la solution la plus rapide serait de porter d'abord MENTOR (si la taille de la machine le permet) plutôt que de reprogrammer entièrement EDME.

5.3 L'expérimentation

L'expérience, décrite dans (HOC,84) vise à déterminer les stratégies employées par les programmeurs lorsqu'ils *inventent* un programme. Techniquement, elle s'est déroulée en trois phases :

1. mise au point du protocole d'expérimentation
2. travail des sujets avec recueil et stockage d'une image de leurs sessions
3. exploitation des recueils.

Nous présenterons ici essentiellement le second point, les deux autres étant hors du cadre de notre étude.

Du protocole, deux éléments sont à retenir : les sujets sont tous des programmeurs professionnels ayant suivi une scolarité à l'IUT de Nancy (donc une formation au langage MEDEE et à la méthode déductive) et les problèmes ont donné lieu à une étude préalable sur l'approche qu'adoptent les sujets a priori (HOC,83). Sur le plan du test d'EDME, le choix des sujets est intéressant car il élimine, dans une certaine mesure, les

problèmes d'utilisation d'EDME liés à la syntaxe et donc permet de mettre plus rapidement en évidence les problèmes liés à l'implantation (bogues) ou aux choix (erreurs de conception, insuffisances, ...).

Lors de la définition du recueil des données, il a été décidé de conserver toutes les entrées de l'utilisateur augmentées de l'heure à laquelle chacune a été frappée. Techniquement, la réalisation a été extrêmement simple; en effet, le système d'entrée-sortie de Multics offre la possibilité de positionner un **audit** : c'est à dire un *observateur* invisible qui met dans un fichier tous les caractères qui transitent par un canal d'entrée-sortie.

Incidemment, ce recueil nous a fourni un outil de mise au point très efficace. Un simple filtrage du fichier brut par un processeur **d'audit** permet de reconstituer un fichier qui, pris comme flôt d'entrée d'EDME, rejoue intégralement une session. Il est de ce fait possible de reconstituer exactement un état d'algorithme et de système. Dès lors la localisation des bogues a été grandement simplifiée et leur correction toujours très rapide. Il est certain que si nous avions à construire un outil interactif sur un système autre que Multics, nous développerions un mécanisme d'audit pour la phase de tests, tant cette expérience nous a convaincu de son utilité.

5.3.1 Les premières constatations

L'analyse des recueils de sessions est en cours et les observations que nous formulons ici concernent uniquement l'aspect utilisation d'EDME.

Les points positifs :

D'une façon générale, le système a été vite appris, en quelques heures, malgré la faiblesse de la documentation et l'absence d'aides *en ligne* (compensées il est vrai par la présence d'un moniteur) Ceci est à comparer au temps que nous avons passé à apprendre MENTOR ou EMACS : de l'ordre de deux mois ! Le choix de situer les commandes sur un plan *logique* plutôt que sur un plan *représentation* est donc payant; de même que l'intégration des contrôles qui accélère l'apprentissage de la syntaxe.

Concernant les bogues : leur nombre a été faible, leur correction rapide et tous les algorithmes ont pu être menés à terme. Il est intéressant de noter que les erreurs se sont toujours manifestées dans des configurations relativement complexes qui n'étaient pas apparues lors des réflexions sur les différentes commandes et fonctions de contrôle. Il faut aussi signaler qu'un certain nombre de bogues nécessitent pour être corrigées un profond remaniement de la grammaire vers une plus grande simplicité et une meilleure «orthogonalité» des constructions. Nous avons donc été amenés dans un certain nombre de cas à fixer de nouvelles contraintes comme par exemple : l'obligation de mettre les conditions d'arrêt sous forme d'un booléen unique, ou encore de ne pas utiliser la possibilité offerte par la syntaxe de remplacer un nom de module par une définition simple dans une définition structurée.

Il a toujours été possible de contourner les difficultés en construisant l'algorithme un peu différemment, y compris dans les cas où EDME commettait une erreur. Ceci indique que l'éditeur est suffisamment robuste vis à vis des erreurs logiques dans les commandes et, qu'en particulier, l'auto-contrôle de la cohérence et l'élimination des états *instables* de la structure éditée sont réellement utiles à ce type d'outil. Bien évidemment, les fautes de programmations (les vraies bogues) sont pour l'instant irrécupérables !

Les points mitigés :

La répartition des commandes et les informations affichées entre et pour chaque couche ne sont pas suffisamment claires. Il faut reconcevoir cette partie, en se concentrant plutôt sur les couches Environnement et Algorithme qui ont été un peu délaissées au profit de la couche Module.

EDME s'est souvent arrêté en catastrophe sur un problème de limitation de l'espace de travail virtuel. Ce problème est essentiellement lié à la gestion de l'espace de travail de Multics qui n'est jamais réinitialisé au cours d'une session. Il suggère tout de même que notre maquette utilise beaucoup de mémoire et qu'il faudra étudier cette question pour installer EDME sur un poste de travail personnel.

Les points négatifs :

La gestion de l'affichage et des messages d'erreur. Ce point a beaucoup gêné les sujets et a probablement allongé la phase d'apprentissage.

L'absence d'un *bon* système d'assistance en-ligne.

L'absence de mécanisme de récupération d'erreurs

La grande difficulté, voire l'impossibilité, de modifier un algorithme. Ce point est certainement le plus négatif car il empêche les démarches par essai-erreur, gêne certaines constructions purement incrémentales et déductives comme par exemple la détermination petit à petit des champs d'une structure, et laisse supposer qu'en l'état actuel, un gros algorithme ne pourrait certainement pas être construit. Il faut noter que ce point provient pour beaucoup de la méthode que nous employons car celle-ci insiste plus sur la création et ne traite pas beaucoup de la modification d'algorithmes.

Il ne faut pas voir dans l'énumération des aspects négatifs ci-dessus un échec d'EDME mais bien plus une mise en évidence expérimentale des outils et caractéristiques nécessaires à un éditeur "intelligent" *réellement* efficace. Nous avons pu mettre ces points en évidence grâce au fait que les concepts et les idées qui ont présidé à la réalisation actuelle se sont révélés pertinents (structure unique pour l'utilisateur, dialogue, multi-fenêtrage, simplicité, incrémentalité, contrôles intégrés, ...).

Nous estimons également que le but de faire d'EDME une plate-forme expérimentale, tant pour l'étude de la programmation que pour la conception et la réalisation d'outils, est atteint car de l'observation de ces différents points, principalement des négatifs, nous avons tiré les critiques et proposé les améliorations décrites au chapitre 7.

Chapitre 6

La mise au point des algorithmes

Lorsque nous avons présenté l'architecture de Malday (cf chapitre 3.2), nous avons cité un interprète. Il nous semble en effet indispensable qu'un environnement de programmation dispose d'un outil permettant une évaluation immédiate d'un fragment de texte. L'interprète de MEDEE n'est pas encore implanté. Nous donnons dans ce chapitre une description de cet outil sous l'angle des choix de conception et d'études préalables à une réalisation.

Parmi les solutions au problème de la mise au point d'un algorithme MEDEE, il est possible d'envisager un mécanisme permettant de prouver formellement sa correction. Nous avons rejeté ce choix en estimant qu'un mécanisme d'interprétation est plus accessible à un utilisateur novice. La première section de ce chapitre justifie ce choix.

La seconde section propose une définition d'un interprète. Conformément à la démarche suivie jusqu'ici, nous présentons les fonctionnalités de l'outil perçues par l'utilisateur. Puis nous proposons une description formelle de l'interprète sous la forme d'algorithmes MEDEE. Cette définition *méta-circulaire* est intéressante en ce qu'elle donne une *spécification opérationnelle* d'un interprète et qu'elle permet d'étudier des extensions du langage.

L'interprète proposé est destiné à être intégré à l'éditeur, la troisième section présente donc les aspects de *coopération* entre les deux outils.

6.1 Preuve et mise au point

Il peut sembler paradoxal de parler de mise au point d'un algorithme : en effet, un algorithme peut être considéré comme un système d'équations exprimant les relations entre les objets, sur lequel une preuve formelle peut être conduite. Cette approche est particulièrement nette avec LUCID (AHS,77) qui est un langage d'expression d'algorithmes en même temps qu'un système formel doté de règles d'inférences appropriées. La finalité de MEDEE est légèrement différente : le langage doit être un support pour la construction d'algorithmes.

Examinons ce que nécessite une preuve.

1. Il faut une spécification formelle du résultat souhaité. EDME est conçu pour des utilisateurs non informaticiens. Il est donc peu probable qu'un algorithme écrit avec EDME soit dérivé d'une

spécification formelle, d'autant plus qu'actuellement nous ne sommes pas en mesure de proposer un outil d'aide à la construction de spécifications.

2. Il faut mettre en oeuvre des techniques complexes différentes de celles de la programmation. Par exemple, la preuve d'une itération nécessite la construction d'un invariant et d'une fonction décroissante à valeur dans un ensemble muni d'un ordre «bien fondé», ces opérations sont ardues et souvent hors de portée d'un débutant.
3. Si le succès de la preuve doit être le résultat final de l'étape algorithmique, l'échec de la preuve, fréquent lors de la construction de l'algorithme, donne très peu d'indications sur ses causes.

Pour ces trois raisons, nous préférons offrir au programmeur la possibilité de tester son algorithme, au sens de tester un programme, c'est à dire : effectuer des calculs sur des jeux d'essais. La perte de rigueur inhérente à cette démarche nous semble largement compensée par l'aide apportée à l'utilisateur. Il convient de noter qu'apparaît une difficulté : nous avons pris soin, lors de la conception d'EDME, d'éliminer les contraintes d'exécution et de privilégier une vision statique de l'algorithme, il faut donc que l'interprétation reste cohérente avec ce choix.

6.2 L'interprétation

Le vocabulaire que nous employons dans ce chapitre nous est particulier. Nous précisons donc ici le sens que nous attribuons à quelques termes.

objet. C'est le terme générique que nous employons pour désigner les entités définies. Un objet est désigné par un nom (l'identificateur utilisé dans l'algorithme). Implicitement, tout objet est une SUITE, le type qui lui est associé dans le lexique est le type des valeurs des éléments atomiques de la suite.

occurrence. C'est un couple <identificateur,indices> qui permet de désigner un élément particulier de la suite associée à l'objet désigné par l'identificateur. On peut noter que l'élément désigné peut lui-même être une suite dans le cas où l'objet est défini par plusieurs niveaux de récurrence. Dans le cas où l'élément désigné par une occurrence est une sous-suite, la valeur associée est celle de l'occurrence désignant le dernier élément de cette sous-suite.

historique. C'est la suite de toutes les valeurs calculées d'un objet, lors d'une exécution (partielle ou totale) de l'algorithme.

6.2.1 Le point de vue de l'utilisateur

Rappelons tout d'abord que MEDEE ne doit pas apparaître à l'utilisateur comme un langage de

programmation habituel avec des variables. MEDEE est un langage de spécification de calculs dans lequel les notions importantes sont celles *d'objets* et de *relations entre objets*, EDME est conçu pour faciliter la manipulation de ces concepts. L'essentiel du travail d'EDME est donc d'assurer la correction de la spécification sur le plan *définitionnel* : c'est à dire que l'algorithme spécifie un calcul valide. L'utilisateur a évidemment besoin de vérifier aussi la correction *opérationnelle* de son texte : c'est à dire que l'application de la spécification à des valeurs réelles fournit bien le résultat espéré. L'outil que nous proposons, INTERMEDE (pour INTERprète MEDEe), doit permettre la *validation expérimentale* du texte.

Il est délicat de définir a priori ce dont a besoin l'utilisateur cible (un débutant) d'EDME; nous voyons cinq fonctions utiles :

1. Le calcul d'un objet. C'est évidemment la fonction essentielle d'un interprète. Le calcul peut être lancé à tout moment depuis EDME, sans que l'algorithme soit complet ni ordonnancé. INTERMEDE considère toute occurrence non définie comme une donnée dont la valeur est fournie par l'utilisateur, de même dans le cas de modules ou de procédures non définis, l'utilisateur donne directement les valeurs des résultats. Cette fonction n'affiche que la dernière valeur de l'objet calculé, les autres valeurs (les éléments de la suite) sont visualisables indépendamment.
2. Le calcul d'une expression. Il est fréquent, surtout avec les expressions conditionnelles, que l'on souhaite vérifier qu'une expression complexe correspond bien à celle que l'on a à l'esprit, INTERMEDE permet donc de calculer un résultat à partir des valeurs des objets apparaissant dans l'expression qui sont toutes données par l'utilisateur.
3. La visualisation d'un historique. Ces valeurs sont calculées avec la fonction (1) mais n'apparaissent pas lors du calcul. Nous avons choisi cette solution plutôt qu'une trace classique car l'ordre de calcul des éléments par INTERMEDE peut être quelconque et le caractère fugitif d'une trace ne permet pas toujours de bien analyser ce qui se passe. L'intérêt de cette fonction est surtout de vérifier les récurrences.
4. La visualisation du graphe de dépendance des objets. Cette fonction est intermédiaire entre EDME est INTERMEDE car elle visualise les relations et les contraintes d'ordonnancement.
5. La visualisation de l'expression instanciée qui a servi à calculer un objet. Cette fonction doit permettre de reconstituer l'historique local d'un calcul particulier.

Nous avons pris soin dans la définition d'INTERMEDE d'éliminer les problèmes de dynamique des calculs (ordonnancement des définitions, ordre d'entrée des données, ...). INTERMEDE doit être considéré comme un résolveur de systèmes d'équations (c'est ainsi qu'est vu un algorithme MEDEE) qui fournit la solution globale du système.

6.2.2 INTERMEDE en MEDEE

Nous présentons ci-dessous une description d'INTERMEDE écrite en MEDEE. Nous avons pris la liberté d'étendre quelque peu le langage en admettant des types non-spécifiés pour les objets dans l'algorithme, les types introduits étant décrits par ailleurs. Nous justifions ceci par le fait que MEDEE est un langage de bas niveau, particulièrement en ce qui concerne la description des types, et l'introduction de présentations inspirées des types abstraits nous semble une évolution souhaitable de MEDEE. Nous nous sommes surtout attaché à conserver l'esprit du langage.

Tout l'interprète n'est pas spécifié ici, seule la partie centrale qui décrit comment sont calculés les objets et enchaînés les calculs apparaît. Les autres composantes d'INTERMEDE sont décrites dans les sections suivantes.

Afin de faciliter la compréhension des algorithmes, nous donnons ici les idées générales qui nous ont guidé.

Pour calculer une occurrence il faut au préalable avoir calculé toutes les valeurs des occurrences dont elle dépend. Le mécanisme le plus immédiat pour implanter cette caractéristique est *l'appel par nécessité* que nous avons donc utilisé.

Il faut conserver toutes les valeurs d'un objet (historique). Ceci est rendu nécessaire par l'emploi des appels par nécessité et les fonctions de visualisation des valeurs proposées à l'utilisateur.

Tout objet utilisé mais non défini (ou imparfaitement défini comme dans l'expression

```
a: jqa arrêt repeter MOD
```

où MOD est vide), est une donnée (ceci est cohérent avec la démarche déductive).

Le calcul d'une suite (itération) se compose en fait d'un double calcul : le calcul de l'objet itéré et le calcul d'une *suite guide*. L'arrêt du calcul correspond au moment où la suite guide atteint une certaine valeur dépendante de la forme syntaxique de la définition itérative employée. Ceci permet de mettre toutes les itérations sous une forme "canonique" :

```
a: INIT jqa Arrêt-a repeter MOD.
```

Le calcul d'un objet conditionnel nécessite l'évaluation de toutes les conditions. Ceci est obligatoire si l'on veut conserver les propriétés d'unicité de définition et d'indépendance d'un ordre d'évaluation. Ceci implique qu'un calcul validant simultanément plusieurs conditions ne peut se poursuivre et donc que l'algorithme est faux.

```
fonction CALCULE : TOccurrence x TAlgo x THistorique  --> THistorique

résultat : N-Historique

N-Historique =E-Range(Historique,Occurrence,Val)

Historique, Occurrence : paramètres

Val : si Def-Par-Simple(Obj)  alors Val = Eval-Simple (Occurrence,Obj,Historique,Algo)
      si Def-Par-Iter (Obj)   alors Val = Eval-Iter (Occurrence,Obj,Historique,Algo)
      si Def-Par-Cond(Obj)    alors Val = Eval-Cond (Occurrence,Obj,Historique,Algo)
      si Def-Par-Donnee(Obj)  alors Val = Utilisateur (Occurrence)
      si Non-Defini (Obj)     alors Val = Utilisateur (Occurrence)

 Algo : paramètre

 Obj = Acces-Objet (Algo,Occurrence)


-------------------------------------------------------------------------------
  Fin de Calcule
-------------------------------------------------------------------------------
```



```
-------------------------------------------------------------------------------
  Définition informelle des Objets et Types

 Occurrence ( TOccurrence = produit-cartésien de
                          ( Id  ( TIdent )
                            L-Ind ( liste de ( TIndices ))))
 Correspond à un identificateur muni de la liste des valeurs d'indices pour
lesquels on calcule la valeur

 Historique ( THistorique = table de
                           ( Id (TIdent)
                             L-Val ( liste de ( TValeur))))
 Historique représente la "mémoire" de l'interprète dans laquelle sont rangés,
associés à chaque identificateur, les historiques.

 N-Historique ( THistorique )

 Algo (TAlgo = Ensemble de ( TObjet ))
  C'est l'ensemble des objets et des "équations". Il s'agit en fait d'un point
de vue sur l'algorithme édité qui est plus adapté à l'interprétation.

 Obj ( TObjet)
Structure construite pour une définition et un objet en un point particulier de
l'algorithme. Il s'agit du point de vue de l'interprète sur une définition
particulière.

 Val ( TValeur)

 Les types TValeur et TIndices sont considérés comme préexistants.
 Le type TObjet sera construit peu à peu, au cours de la "spécification"

-------------------------------------------------------------------------------

  Definition Informelle des fonctions introduites (avec le profil)

Def-Par-Simple  TObjet --> Booléen
Def-Par-Iter    TObjet --> Booléen
Def-Par-Cond    TObjet --> Booléen  Fonctions du type TObjet. Déterminent quel
Non-Défini      TObjet --> Booléen  est le type de définition définissant
Def-Par-Donnee  TObjet --> Booléen  l'objet.


Eval-Simple     TOccurrence x TObjet x THistorique x TAlgo --> TValeur
Eval-Iter       TOccurrence x TObjet x THistorique x TAlgo --> TValeur
Eval-Cond       TOccurrence x TObjet x THistorique x TAlgo --> TValeur
  Les trois formes d'évaluateurs, une par type de définition

Utilisateur     TOccurrence  --> TValeur   Fonction qui demande une Valeur à
                                           l'utilisateur. C'est la procédure
                                           d'entrée de l'interface utilisateur.

Acces-Objet   TAlgo x TOccurrence --> TObjet  Fonction qui pour un algorithme
                                              donné et une occurrence, recherche la
                                               définition et l'Objet associé.


-------------------------------------------------------------------------------
fin de calcule
-------------------------------------------------------------------------------
```

```
fonction  Eval-Simple  : TOccurrence x TObjet x THistorique x TAlgo --> TValeur

 résultat = Val

 Val = Expr-Eval( Expr, L-Val-Id )

 Expr = Obj-Expr(Objet)

 Objet : paramètre

 L-Val-Id : INIASSOC
            pour I de 1 a long(Obj-Lpatron(Objet))
               repeter ASSOC

 L-P = Obj-Lpatron (Objet)

 Historique, Algo, Occurrence : paramètre

=============================================================
ASSOC
=============================================================

 L-Val-Id = Dajoute(@L-Val-Id, Val-Id)

 Val-Id = Ident de L-P[I] , Valint

 Valint = Cherche-Valeur ( Occint, Historique, Algo)

 Occint = Ident de L-P[I] , L-ind

 L-ind = Calcule-Indices(L-Ind de Occurrence , patron de L-P[I])

=============================================================
INIASSOC
=============================================================

 L-Val-Id = vide


-------------------------------------------------------------------------
fin de Eval-Simple
-------------------------------------------------------------------------
```



```
-------------------------------------------------------------------------
    Définition informelle des objets et types de Eval-Simple

Val (TValeur)  valeur calculée

Expr (Abs-Tree) expression syntaxique sous forme d'arbre abstrait

L-Val-Id ( TLV= liste de ( produit-cartésien de
                           ( Id    (TIdent)
                             Val   ( TValeur) )))
 Table associant à chaque identificateur apparaissant dans la définition sa
valeur (valeur de l'objet pour une valeur d'indices particulière)

L-P (Tgraph= liste de ( produit-cartésien de
                        ( Ident (TIdent)
                          patron ( liste de (Tsymb)))))
 Cette liste définit le graphe de dépendance local en donnant pour chaque
objet utilisé la relation entre ses valeurs d'indice et celles de l'objet
calculé. Tsymb est un type à trois valeurs : = pour l'égalité entre les indices
                                             - pour une différence d'un
                                             * pour prendre "le dernier"

Valint (TValeur)  intermédiaire

Objet, Historique, Algo , Occurrence,: comme pour CALCULE


-------------------------------------------------------------------------

 Définition informelle et profil des fonctions introduites

Obj-Expr       TObjet --> Abs-Tree

Obj-Lpatron    TObjet --> Tgraphe
   ces deux fonctions sont des accès sur le type TObjet

Expr-Eval       Abs-Tree x TLV --> TValeur
  Calcul d'une expression algébrique donnée par l'arbre abstrait. Les valeurs
utilisées sont données par la table L-Val-Id

Dajoute        Liste x Elt --> Liste
 Fonction générique qui ajoute un élément en FIN de liste

Cherche-Valeur   TOccurrence x Historique x Algo --> TValeur
 Renvoie la valeur associée à une Occurrence, si la valeur a déjà été calculée,
c'est un simple accès dans Historique, sinon, lance récursivement le calcul de
Occurrence

 Calcule-Indices   TLI x TLPat --> TLI
                    avec TLI = liste de (TIndices)
                         TLPat = liste de (Tsymb)
 Détermine les valeurs d'indices associées à une liste de valeurs et un patron

-------------------------------------------------------------------------
fin de eval-simple
-------------------------------------------------------------------------
```

```
fonction Eval-Iter   : TOccurrence x TObjet x  THistorique x TAlgo --> TValeur

résultat = Val

Val = E-Acces (N-Historique, Occurrence)

Historique : paramètre

 L-Ind-Max, N-Historique : INISUITE
                      jqa ArretGuide  repeter  SUITE

 Obj, Algo, Occurrence : paramètre

================================================================
SUITE
================================================================

L-Ind-Max = Incr-Dern(@L-Ind-Max)

ArretGuide = ( Valguide = Obj-Valarret(Obj) )

Valguide = Cherche-Valeur(Occguide,Historique,Algo)

Occguide = Obj-Idarret(Obj), L-Ind-Max

N-Historique = Calcule(Occiter, Algo, @N-Historique)

Occiter = Id de Occurrence , L-Ind-Max


================================================================
INISUITE
================================================================

L-Ind-Max = Ajoute(L-Ind de Occurrence , 0)

N-Historique = Historique

________________________________________________________________
fin de eval-iter
________________________________________________________________
```



```
________________________________________________________________
 Définitions Informelles des objets et types de Eval-Iter

Val, Obj, Algo, Historique, Occurrence : comme pour Calcule

L-Ind-Max (liste de (TIndices)) : liste des valeurs d'indices de l'objet
             itéré, elle a un élément de plus que la liste d'indices de
             l'objet passé en paramètre.

 N-Historique (THistorique)   : suite des Historiques calculés pendant l'itération

 ArretGuide (Booléen)  : suite de Booléens dont le dernier terme vaut VRAI.

 Valguide (TValeur)  : Valeur de la "suite Guide" de l'itération

 Occguide (TOccurrence) :
 Occiter  (TOccurrence) : intermédiaires de calcul


________________________________________________________________
 Définition informelle et profil des fonctions introduites

 E-Acces          Historique x TOccurrence --> TValeur

    Donne la valeur rangée pour une occurrence donnée

Incr-Dern         Liste de (TIndice) --> Liste de (TIndice)

   ajoute UN à la valeur du dernier indice de la liste

Obj-Valarret      TObjet --> TValeur

   Donne la valeur extrême (i.e. la dernière) de la suite guide d'une itération

Obj-Idarret       TObjet --> TIdent

   Donne le nom de la suite guide.

Note : La suite guide est un objet particularisé qui permet de déterminer la
longueur d'une itération.

________________________________________________________________
fin de Eval-Iter
________________________________________________________________
```

```
fonction Eval-Cond   : TOccurrence x TObjet x THistorique x TAlgo --> TValeur

résultat = Val

Val = E-Acces ( N-Historique, Occurrence)

 N-Historique : si  calculable
                    alors TRANSFERT
                    sinon N-Historique= Historique

 collision, k : INIBRANCHE
                    pour I de 1 a Obj-NBcond(Obj)
                    repeter TESTBRANCHE

k1 : si k=0
          alors k1=Obj-NBcond(Obj) + 1
          sinon k1 = k

calculable : si (collision=faux) et (k=0)
                 alors calculable = vrai
                 sinon calculable = faux

Algo, Historique, Obj , Occurrence : paramètre

=============================================================
TESTBRANCHE  calcul de toutes les conditions pour savoir
             quelle branche choisir ou éventuellement sortir
             en erreur si deux conditions sont valides
             simultanément
=============================================================

 Historique-Int = calcule (Occond,@Historique-Int,Algo)

 Occond = L-Cond[I] , L-Ind de Occurrence

 L-Cond = Obj-LCond (Obj)

 collision : si E-Acces(Historique-Int,Occond)=vrai et k=0
               alors  collision=faux
               sinon  collision=@collision

 k : si E-Acces(Historique-Int,Occond)=vrai
               alors  k=I
               sinon k= @k

============================================================
INIBRANCHE
============================================================

k = 0

collision = faux

Historique-Int = Historique

============================================================
TRANSFERT    On calcule l'objet défini dans le module
             conditionnel sélectionne (le k1-ieme)
============================================================

 N-Historique = calcule(OccKieme,Algo,Historique)

 OccKieme = LKOcc[k1] , L-Ind de Occurrence

 LKOcc = Obj-L-Id (Obj)

------------------------------------------------------------
fin de Eval-Cond
------------------------------------------------------------
```



```
------------------------------------------------------------
   Définitions informelles et type des objets de Eval-Cond

Val, Historique, Occurrence, Algo : comme pour les autres fonctions

N-Historique (THistorique)  : état résultant

calculable (Booléen) : vrai si une et une seule condition est validée

collision (Booléen) : vrai si DEUX conditions explicites sont vraies
                      simultanément

k1  (Entier)  : numéro de la branche conditionnelle choisie (nécessaire pour
                séparer la clause sinon)
k   (Entier)  : numéro de la condition explicite vraie, si il vaut 0, c'est
                la clause sinon qui est validée

Occond   (TOccurrence)  : intermédiaire, désigne l'objet booléen représentant
                          une condition

Historique-Int  (THistorique) : intermédiaire

L-Cond  (TL-Id=liste de (TIdent)) : liste des objets booléens représentant chaque
                                    condition


------------------------------------------------------------
 Définition informelle et profil des fonctions introduites

 Obj-LCond :   TObjet --> TL-Id

      donne la liste des conditions attachées à un objet conditionnel

Obj-NBcond :   TObjet --> Entier

      donne le nombre de clauses explicites

Obj-L-Id   :   TObjet --> TL-Id

      donne la liste des noms que possède l'objet conditionnel dans chaque
      module. Il est obligatoire de renommer cet objet si on veut conserver
      l'unicité de définition à une profondeur itérative donnée.

------------------------------------------------------------
fin de Eval-cond
------------------------------------------------------------
```

```
fonction Cherche-Valeur   :  TOccurrence x THistorique x TAlgo --> TValeur

résultat = Val

Val = E-Acces(N-Historique, Occurrence)

N-Historique : si E-Acces(Historique,Occurrence) = vide
               alors N-Historique= Calcule(Occurrence,Historique,Algo)
               sinon N-Historique= Historique

Occurrence, Algo, Historique : paramètre

-------------------------------------------------------------------

fin de Cherche-valeur

-------------------------------------------------------------------
```


6.2.3 L'explicitation des indices

Dans les algorithmes précédents, les listes d'indices jouent un grand rôle dans la définition des calculs du fait du caractère itératif de MEDEE et du type des objets manipulés : des suites. La syntaxe du langage a été définie de façon à rendre implicite l'indiçage des objets, ce qui facilite et simplifie la construction de l'algorithme mais oblige à recalculer les indices, ou plus exactement leur nombre, avant toute interprétation. Cette section présente les règles et les fonctions nécessaires à cette explicitation.

La notion clé est celle de **profondeur itérative** d'un objet. Cette profondeur correspond au nombre d'indices dont dépend un terme d'une suite; elle reflète le fait que les objets MEDEE peuvent être des suites à plusieurs niveaux.

La profondeur d'un objet se calcule à partir de celle des modules. Ceux-ci obéissent aux règles suivantes :

1. la profondeur d'un module itératif est celle du module père augmentée de un
2. la profondeur de tout autre module est celle du module père
3. la profondeur du module principal est zéro

Les règles pour les objets en découlent :

1. la profondeur d'un objet défini (i.e. en partie gauche d'une définition) est celle du module englobant
2. la profondeur d'un objet utilisé (i.e. en partie droite d'une définition) est la profondeur du premier module aïeul en remontant la hiérarchie des modules dans lequel l'objet est défini
3. la profondeur d'un objet récurrent défini dans un module d'initialisation est augmentée de un; le dernier indice a la valeur zéro.

Le calcul peut s'exprimer à l'aide des fonctions suivantes :

Appelons pi(M) la Profondeur Itérative du module M

```
pi(M) = si  Est-principal(M)  alors 0
          sinon
          si Type-itératif(M)         alors pi(Père(M)) + 1
          si Type-conditionnel(M)     alors pi(Père(M))
          si Type-initialisation(M)   alors pi(Père(M))
```

où Père est la fonction qui associe à un module le module qui l'a introduit.

```
MM = Père(M) ssi M ∈ Lexique-Mod(MM)
```

où Lexique-Mod est la fonction qui donne les noms des modules introduits dans un module donné.

Appelons pio(Obj,Def) la Profondeur Itérative de l'Objet Obj dans la définition Def. Soit M le module dans lequel Def est située

```
pio(Obj,Def) = si Défini(Obj,Def)
                 alors si Type-initialisation(M)
                         alors pi(M)+1
                         sinon pi(M)
               si Utilisé(Obj,Def)
                  alors pi(Dern-Def(Obj,M))

Dern-Def(Obj,M)= si Est-défini-dans(Obj,M)
                    alors M
                    sinon Dern-Def(Obj,Père(M))
```

Les fonctions préfixées par Type– indiquent la catégorie du module paramètre; cette catégorie est fixée par la définition qui introduit le module.

Les fonctions Défini et Utilisé indiquent si, pour une définition donnée, un objet est en partie gauche ou droite.

La fonction Est-défini-dans indique si un objet donné se trouve en partie gauche d'une des définitions du module. Si on veut faire ce calcul sur un algorithme non terminé, il faut considérer aussi la liste des *objets à définir* associée au module.

Le lecteur intéressé trouvera dans (LES,78) une sémantique d'un MINI-MEDEE où sont introduites et développées ces notions de profondeurs itératives. La figure 11 présente des modules de l'algorithme du Chapitre 1 dans lesquels les indices sont explicités.

module EMPAQUETTE

$sflot_{<i>} = sflot_{<i-1>}$ suivide $paquet_{<i>}$

$paquet_{<i>} = impr(\ bloc_{<i>}\ )$

$stop_{<i>} = extra\ (\ bloc_{<i>}\ ,\ long(bloc_{<i>})\ ,\ 1) = Eot_{<i>}$

$bloc_{<i>}\ ,\ finchaine_{<i>} = INIBLOC$
jqa $stopbloc_{<i,j>}$
répéter BLOC
avec $Eot_{<i>}$

$Eot_{<i>} = donnée$

module COUPE

$bloc_{<i,j>} = bloc_{<i,j-1>}\ !\ debchaine_{<i,j>}$

$debchaine_{<i,j>} = extra(\ enchaine_{<i,j>},\ 1\ ,\ lmaxbloc_{<0>} - long(bloc_{<i,j-1>})\ )$

module INIBLOC

$BLOC_{<i,0>} = finchaine_{<i-1>}$

$finchaine_{<i,0>} = finchaine_{<i-1>}$

Figure 11. algorithme avec les indices explicités

6.2.4 Le calcul des expressions algébriques

La résolution de ce problème est triviale sur une structure d'arbre abstrait : en effet une simple traversée de l'arborescence effectue ce calcul. Nous avons donc essayé de définir un mécanisme qui permette une évolution aisée du langage, une extension des types de base et éventuellement l'adjonction d'opérations pré-compilées sur des types de plus haut niveau, et qui soit programmable dans le cadre de MENTOR (donc en utilisant essentiellement PASCAL).

Afin de faciliter l'évolutivité, il ne faut pas avoir à modifier les programmes à chaque «retouche» sur les opérateurs, en particulier lorsqu'on en ajoute un nouveau. Il faut donc que le parcours de l'arbre soit *guidé par les données*.

Le mécanisme que nous avons retenu consiste à associer à chaque opérateur de la grammaire susceptible de se trouver dans une expression algébrique une procédure PASCAL réalisant le calcul associé. Le nom de la procédure est composé de la concaténation du nom de l'opérateur et du nom du type abstrait dont il fait partie. Ceci est rendu obligatoire par la surcharge des opérateurs de MEDEE, essentiellement en ce qui concerne les opérateurs à résultat booléen. Le parcours de l'arbre est implicitement réalisé par l'appel d'une fonction qui prend comme paramètre un arbre abstrait et un nom de type, construit le nom de la procédure PASCAL et active celle-ci.

L'activation se fait par le passage du nom construit à l'interprète de commandes du système opératoire (dans l'essai réalisé : Multics) qui active la procédure désignée. Le passage des valeurs et des paramètres s'effectue par une zone de mémoire commune. Le lecteur intéressé trouvera le listage des programmes implantant ce mécanisme dans les annexes.

Ce mécanisme est assez lourd et peu efficace mais il fonctionne correctement. Il faut noter qu'un mécanisme interprétatif de ce type est toujours à programmer dès que l'on veut avoir une certaine extensibilité avec des langages compilés.

6.3 La préparation de l'interprétation

La définition d'INTERMEDE est simple car elle ne comporte pas de traitement d'erreurs. Ce choix est volontaire car :

1. les mécanismes mis en oeuvre par les erreurs sont des traitements d'exception, domaine pour lequel MEDEE ne possède pas de constructions appropriées.

2. mais surtout peu d'erreurs peuvent stopper l'exécution du fait du nombre élevé de contrôles effectués auparavant (correction syntaxique, correction contextuelle, ...). Les erreurs les plus fréquentes avec les langages fortement typés comme PASCAL sont des accès en dehors de la zone où la mémoire est définie (pointeurs mal entretenus et débordement d'indices dans les tableaux). La notion de pointeur n'existe pas en MEDEE, donc une source d'erreurs est éliminée. Pour ce qui concerne les tableaux, la technique la plus sûre est de contrôler qu'à chaque accès la valeur de l'indice appartient à l'intervalle de variation.

Ce traitement préalable des algorithmes est important et une certaine partie des traitements liés à l'interprétation peuvent être effectués sitôt une définition intégrée à l'algorithme. Parmi ces traitements, citons :

- le calcul de la structure OBJET :
  - le type de définition
  - la liste de dépendance avec les patrons d'indices associés
  - les différents renommages (objets conditionels)
  - détermination de la suite guide (itération)
- les *modifications* de l'algorithme :
  - mise sous forme canonique des itérations
  - renommage des objets conditionnels
- les calculs de profondeur itérative et leur propagation (cas des modules introduits, ...)

Il est intéressant de séparer dans le temps ce qui concerne la construction des structures sur lesquelles travaille l'interprète, du calcul effectif, d'autant que les informations calculées sont réutilisables, principalement dans des fonctions d'aide à l'utilisateur permettant d'avoir des points de vues autres que syntaxiques sur le texte : par exemple, le graphe de dépendance des objets, ou encore la structure des itérations (suite guide), ...

Chapitre 7

Les Premières Critiques

Nous présentons dans ce chapitre quelques aspects d'EDME sur lesquels d'importants travaux doivent être entrepris. Plutôt que de rechercher à définir a priori des extensions d'EDME, nous avons préféré conserver le caractère expérimental de notre démarche. Ainsi le choix des points présentés ci-après découle de l'analyse des premières utilisations d'EDME, et en particulier des difficultés éprouvées par les novices. Nous renvoyons le lecteur au chapitre 5 section 3 dans lequel sont présentées les observations réalisées lors de l'emploi d'EDME.

7.1 L'amélioration de l'affichage

Nous avons soigneusement conçu l'affichage en insistant sur les critères de lisibilité, d'agrément de lecture et de *compréhensibilité*. De fait, la réalisation actuelle est généralement satisfaisante quoique quelques problèmes puissent apparaître. Par exemple, il arrive qu'un passage à la ligne se produise entre le symbole @ et l'identificateur qu'il qualifie. Ceci est une bogue qui peut être corrigée assez simplement mais, d'une façon générale, la question de définir une présentation *optimale* restera posée.

En effet, pour qui lit des programmes, il apparaît très vite que les formatages (mise en page, typographie, indentation, ...) facilitent plus ou moins la compréhension selon qu'ils sont plus ou moins proches des habitudes et des normes adoptées par le lecteur. Or les outils comme EDME prennent entièrement en charge la présentation des textes. Ce point peut sembler annexe, en fait, nous lui accordons une certaine importance. En rendant la compréhension d'une définition *immédiate*, on augmente la concentration du programmeur sur l'aspect *sémantique* du texte et on diminue la fatigue inhérente à la lecture sur écran.

Des solutions pour adapter la présentation à l'utilisateur sont envisageables. Si nous ne croyons pas trop à la possibilité de paramétrage du décompilateur directement effectué par l'utilisateur à cause de la complexité d'une telle tâche et du haut niveau de technicité qu'elle requiert, nous pensons qu'il serait possible d'offrir un choix entre quelques types de présentation prédéfinis. Il faut alors que ce choix soit modifiable à tout instant pour que le programmeur puisse essayer les diverses alternatives.

Il serait à notre avis intéressant d'utiliser EDME pour *mesurer* les facilités de lecture et de compréhension d'algorithmes en fonction de divers types de formatage et de syntaxe concrète du langage.

Pour conclure sur cet aspect d'affichage, nous voudrions l'élargir au problème de la *lecture* d'un algorithme. Actuellement, l'affichage suit exactement la structuration syntaxique de MEDEE, ce qui interdit par exemple

de mettre côte à côte deux définitions apparaissant dans des modules différents. Il serait certainement très intéressant d'offrir aux utilisateurs plusieurs points de vue lors de la lecture, en particulier, des rapprochements entre objets épars dans le texte. L'affichage ne travaillerait plus alors sur un arbre mais sur des graphes plus complexes. De telles notions ont déjà été étudiées pour des éditeurs de textes (cf les hyper-textes présentés dans (M&VD,82)).

7.2 La gestion des messages

Un système interactif comme EDME est amené à émettre un grand nombre de messages. Nous retrouvons donc tous les problèmes liés aux messages: pertinence, redondances, compréhensibilité, erreurs induites par une autre, espace disponible, enchaînement des messages (un message recouvre le précédent), etc.

La politique de gestion de messages que nous avons adoptée résulte de compromis d'implantation et d'observations sur les messages d'erreur de nos outils habituels. Techniquement les contraintes sont: une faible taille d'affichage (2 à 3 lignes) et l'absence d'interprète de commandes EDME (utilisation de l'interprète MENTOR). La seconde contrainte implique que chaque commande gère elle-même ses messages et l'écran, et que les communications entre commandes sont très limitées.

Un message possède plusieurs caractéristiques:

1. une longueur (d'un mot à une phrase complète).
2. une adaptabilité vis à vis du texte. Son libellé peut être constant ou au contraire intégrer un morceau de texte.
3. une rémanence. Le temps pendant lequel le message peut être consulté peut être quasi-nul (bip sonore), bref (*flash* sur l'écran) ou long (demande d'effacement explicite par l'utilisateur).
4. une pertinence (par opposition à l'absence quand par exemple dans une cascade, les messages sont tous issus de la même erreur et n'apportent que peu d'informations nouvelles).

Pour EDME, nous avons adopté une attitude «conservative»: nous affichons le plus longtemps possible les informations les plus importantes (la notion de *plus important* est ici très subjective). Ainsi, lors d'une entrée de définition, la définition formelle est affichée tant que dure l'analyse, les définitions informelles alors construites n'apparaissent pas; en cas d'erreur, le premier message généré par un des mécanismes de contrôle est affiché, les messages suivants sont ignorés. Les messages et la définition mise dans la fenêtre temporaire ne sont effacés qu'en cas de succès de la commande, de ce fait, en cas d'erreur, le message et la définition reste à l'écran jusqu'à l'activation de la commande suivante.

Ce choix a le mérite de la simplicité, tant pour la réalisation que pour l'utilisation, et s'avère être

raisonnable vis à vis des objectifs d'assistance d'EDME. Toutefois, quelques utilisateurs ont été gênés par deux points:

1. la rémanence de certains messages entre deux commandes «perd» le programmeur qui ne situe plus l'état actuel d'EDME
2. le non affichage des définitions informelles lors d'un **.CRDEF** qui peut être gênant lorsque des objets possèdent des types structurés complexes.

D'autres critiques sont possibles comme par exemple la désignation d'une seule erreur par définition, alors qu'il peut être souhaitable de signaler toutes les erreurs indépendantes, ou le fait que le libellé des messages soit fixe, ce qui peut devenir lassant pour les messages qui reviennent souvent. Enfin, techniquement, il est difficile de maintenir les commandes du fait de l'imbrication du code propre et de celui de gestion des messages.

La solution à ces problèmes passe par la réalisation de plusieurs outils:

Un interprète de commandes propre à EDME, chargé d'activer les outils et de gérer l'écran (nettoyage, réaffichage, ...)

Un *analyseur d'utilisateur*, c'est à dire un système qui construise une image de l'utilisateur reflétant, par exemple, son niveau d'expertise avec MEDEE et EDME, ses stratégies de constructions, ses types d'erreurs fréquentes, ... Un tel outil nous semble indispensable pour personnaliser au mieux les réponses d'EDME et en faire un réel partenaire du programmeur.

Un gestionnaire de messages qui, en fonction du profil de l'utilisateur, de l'état d'EDME, des types de messages, etc, décide de leur affichage, leur rémanence, leur effacement, ...

Des mécanismes d'affichage permettant, par exemple, de faire «rouler» tous les messages relatifs à une définition ou encore de recouvrir temporairement l'écran (*pop-up* messages), ..., et offrant plus de souplesse qu'actuellement dans le dialogue (demande d'affichage pendant l'entrée d'un texte par exemple).

7.3 La gestion des erreurs et des modifications

Un des aspects les plus critiquables d'EDME est la perte de toute l'information entrée lors d'un .CRDEF en cas d'échec de la commande. Ce choix, indispensable pour assurer l'incrémentalité de la construction et pour garantir qu'EDME est dans un état stable entre deux commandes, est actuellement frustrant pour l'utilisateur qui est obligé de frapper plusieurs fois les mêmes textes. Il faut donc un mécanisme permettant de récupérer ces erreurs *au vol*.

Techniquement, plusieurs possibilités se dégagent:

1. Boucler sur l'entrée erronée jusqu'à ce qu'elle soit juste. Cette solution suppose que tout ce qui est entré auparavant est correct mais une erreur détectée en fin de commande peut découler d'une faute dans la définition initiale.
2. Reprendre la commande à son début et faire rééditer les entrées une à une. Cette solution, plus générale que la première, permet de corriger où que soit l'erreur, mais elle est lourde vis à vis de l'erreur fréquente qui est une simple faute de frappe. Une difficulté surgit si la définition formelle initiale est modifiée car alors certaines entrées deviennent sans objet mais certaines peuvent rester pertinentes (cas d'une faute dactylographique dans un identificateur par exemple).
3. Laisser à l'utilisateur le contrôle complet sur la reprise des informations. Cette solution est relativement incompatible avec le désir d'assister le programmeur.

La difficulté de trouver une bonne solution tient au fait que l'information réellement réutilisable est propre à chaque situation. Plus généralement, un problème important d'EDME concerne la modification d'un algorithme, opération dans laquelle il faut essayer de déterminer quelle information est à conserver. En effet, si comme dans l'implantation actuelle, une modification de définition passe par une destruction suivie d'une reconstruction, on peut perdre toute une hiérarchie de modules qu'il faudra rebâtir presque identique. Par exemple la transformation d'une itération **pour** en itération **jusqu'à** est de ce type.

Les difficultés du traitement des modifications sont de plusieurs ordres:

Une modification locale à une définition peut invalider un très grand nombre de définitions. Par exemple le changement de type d'un objet peut rendre incorrectes toutes les définitions qui l'utilisent soit directement, soit indirectement.

Toute modification peut en entraîner d'autres en cascade et il est difficile de circonscrire simplement les points modifiés.

Après une modification, il faut déterminer *l'état* de chaque définition, soit une ou plusieurs des caractérisations suivantes:

- reste correcte
- devient incorrecte vis à vis de la sémantique statique
- devient inutile (par exemple, définit un objet non utilisé par ailleurs)
- devient incohérente vis à vis des intentions du programmeur

Cette dernière éventualité est évidemment non déterminable par EDME qui ignore tout du but de l'utilisateur.

Après une modification, l'algorithme doit être redevenu syntaxiquement, sémantiquement et *méthodologiquement* correct (méthodologiquement signifie ici que tous les objets inutiles ont été éliminés, que les objets à définir ont bien été recalculés (en particulier ceux à initialiser), que les parties **avec** sont à jour, ...). Cette caractéristique est obligatoire si l'on veut conserver une démarche purement incrémentale.

Enfin, toute modification doit pouvoir être *défaite*, i.e. annulée avec retour à l'état antérieur. Cette

caractéristique nous semble indispensable dans le cadre d'un système hautement interactif pour au moins deux raisons: l'utilisateur doit pouvoir adopter, sans risque, une démarche par *essais-erreurs* et il faut aussi prévoir les activations d'une commande par erreur !

Permettre à l'utilisateur de modifier son algorithme revient donc à développer les capacités d'EDME dans quatre directions:

1. L'analyse du texte et en particulier sur l'aspect de la propagation des contraintes, des modifications et des erreurs
2. L'inférence, en particulier en ce qui concerne le calcul et les modifications des types ainsi que l'emploi des objets
3. La conservation des modifications et des actions
4. Un mode de dialogue plus souple

Il est à noter que la résolution de ce problème permettra d'aborder simplement une des autres lacunes actuelles d'EDME: la réutilisation d'algorithmes et de modules déjà créés. En effet, si ces deux problèmes sont bien distincts, ils utiliseront les *mêmes outils*, à savoir l'analyse des objets pertinents et l'inférence (portant ici en plus sur le nom des objets).

7.4 L'inférence

Une des particularités de l'implantation actuelle d'EDME est sa relative «bêtise» : en particulier, il interroge très souvent l'utilisateur sur des points triviaux. Toutefois, il sait quand même déterminer seul quelques informations comme le type des conditions d'arrêt des itérations. En fait, le type des objets nouvellement introduits peut très souvent être directement déduit du contexte dans lequel apparaît l'identificateur. J. Zachary (ZAC,83) a spécifié une procédure d'inférence de types pour MEDEE.

L'idée de base consiste à construire itérativement le type des objets non déclarés en *fusionnant* (*merging*) le(s) type(s) déjà construit(s) pour un objet avec celui requis par un contexte particulier, dépendant en général uniquement de l'opérateur sous lequel apparaît l'identificateur. L'itération s'arrête quand les types construits restent stables par la fusion. Pour ce faire les types sont ordonnés selon une relation spécifiques aux types MEDEE augmentés des types **ANY** (type indéterminé) et **NONE** (type erreur). Cette procédure présente plusieurs intérêts:

1. elle donne des résultats même avec un contexte réduit
2. elle calcule toujours les types simples (y compris **NONE**)

3. elle détecte toutes les erreurs
4. elle donne les caractéristiques minimales que doit posséder le type structuré d'un objet (nombre de dimensions d'un tableau, champs obligatoires d'une structure, ...)

Son implantation est en cours (ZER,84) dans le cadre d'EDME. Nous attendons beaucoup de cette inférence pour améliorer le confort du programmeur. En effet, beaucoup de temps est perdu lors de la construction d'une définition en interrogation sur des types trivialement inférables.

L'inférence portant sur les types est un problème bien formalisé et c'est pourquoi elle est presque implantée dans EDME, mais sont nécessaires à EDME d'autres catégories d'inférences qui nécessitent un effort important de formalisation. Citons en particulier:

Quand deux objets apparaissant dans un algorithme représentent-t-ils «la même chose» ?

Quand un objet est-il constant vis à vis d'une itération?

Ces deux questions sont reliées aux transformations d'algorithmes comme la *fusion d'itération* ou la *remontée des constantes*. Ces opérations sont des réorganisations syntaxiques du texte d'un algorithme destinées à faciliter sa compréhension ou à améliorer son exécution. Elles sont inhérentes à la démarche déductive qui conduit, par exemple, à définir les objets constants par rapport à une itération lors de leur utilisation dans le module itéré, ou encore à définir séparement deux objets récurrents qui se révèleront parcourir le même domaine et utiliser les mêmes objets. Si les règles et les fortes contraintes régissant ces transformations sont relativement aisées à appliquer et à valider à la main, leur automatisation nous semble rester un problème ouvert.

7.5 Les stratégies de construction

Lorsque nous avons présenté le support méthodologique associé à EDME, nous avons montré l'existence d'un certain guide. Cette particularité de la méthode employée permet d'envisager l'automatisation, tout au moins partielle, de l'enchaînement des commandes. La définition de cet enchaînement, et l'outil associé, est ce que nous appelons le *pilote*.

Que devra faire le pilote?

- choisir entre plusieurs actions, c'est à dire mettre en oeuvre une stratégie
  - définir un objet dans le module «courant»
  - changer de module pour définir un autre objet
  - proposer une modification dans les cas d'erreur ou de blocage
  - repasser la main à l'utilisateur
- construire et modifier la stratégie de construction en fonction:
  - de l'utilisateur (novice, expérimenté, ...)
  - du problème (choix défini par l'utilisateur)
  - des erreurs rencontrées
  - de l'état d'avancement du travail
- gérer le dialogue avec le programmeur

La difficulté de la réalisation du pilote tient principalement au fait que l'on connait actuellement assez mal les stratégies mises en oeuvre par les programmeurs et surtout les relations entre stratégie et type de problème à résoudre.

La première utilisation réelle d'EDME aura donc été comme support à une expérience destinée à mettre en évidence des relations entre stratégies, méthodes de travail, ordre d'entrée des définitions (déductif vs prospectif) et catégories de problèmes.

Chapitre 8

Comparaison avec d'autres systèmes

8.1 Les environnements MENTOR

MENTOR (DON,75) et EDME ne sont évidemment pas comparables puisque l'un est l'outil de développement de l'autre, mais les créateurs de MENTOR, en générant des instances de MENTOR pour plusieurs langages, ont réalisé des environnements plus complets. Nous en étudierons trois :

- MENTOR-PASCAL (MEL,81)
- MENTOR-RAPPORT (MEL,84)
- MENTOR-TYPOL (DES,83)

8.1.1 MENTOR-PASCAL

Cet environnement, développé par B. Melese, est formé d'un ensemble de commandes permettant de se déplacer dans un programme PASCAL, de faciliter sa création, de le transformer automatiquement et d'obtenir de la documentation. L'intérêt principal de cet environnement est de montrer les capacités d'extension de MENTOR par des commandes complexes, en même temps, il fournit un excellent exemple de programmation en MENTOL.

Les commandes de déplacement dans l'arbre (**.PROC**, par exemple, qui déplace le pointeur courant sur l'en-tête de la procédure englobante) permettent de simplifier le parcours de la structure éditée. Elles restent néanmoins au niveau syntaxique et ne sont qu'une extension des commandes de base de MENTOR par des commandes fréquemment utilisées. l'utilisateur manipule donc toujours un arbre syntaxique complexe, avec les différentes conséquences que nous avons déjà citées (cf chapitre 2 section 4.1).

Les commandes de création (**.DECVAR**, par exemple, qui insère la déclaration d'une variable dans la partie déclarative du bloc englobant, où que soit le pointeur courant) rendent plus souple l'écriture d'un programme. Il faut noter que le lancement des commandes est entièrement à l'initiative du programmeur. En effet, MENTOR-PASCAL n'effectue pas de contrôles contextuels automatiquement, la vérification du texte sur ce point est donc laissée totalement à la charge du programmeur.

Les commandes de documentation (**.GRAPHE**, par exemple, qui donne le graphe des appels des procédures

d'un programme) sont une aide très appréciable lors de la lecture d'un programme en offrant un point de vue différent sur le texte. Les commandes qui génèrent des commentaires normalisés (auteur et date en tête d'un programme, annotation des terminateurs de blocs par le nom du bloc, ...) traduisent la volonté des concepteurs de MENTOR de promouvoir des normes de programmation. Le point important est, qu'ici encore, l'initiative est laissée au programmeur. EDME est plus directif en imposant et en gérant une documentation plus complète.

Les commandes de transformation (.**RENAME**, par exemple, qui transforme le programme en assurant que tous les objets distincts ont des noms différents) modifient les textes en conservant leur sémantique. Ces fonctions complexes sont surtout utiles pour les programmeurs avertis qui peuvent tirer parti de leur puissance.

L'environnement complet se compose d'une cinquantaine de commandes spécifiques à PASCAL. C'est certainement le point sur lequel la conception de MENTOR−PASCAL nous semble la plus critiquable: en effet, le nombre de commandes est très élevé : 50 pour PASCAL, 50 indépendantes du langage et 20 commandes de base. De ce fait, nombre d'entres elles risquent de ne pas être utilisées par simple méconnaissance de leur existence.

8.1.2 MENTOR−RAPPORT

Le langage Rapport est un formalisme de description de documents. Comme en MENTOR−PASCAL, l'environnement se compose d'un ensemble de commandes permettant de se déplacer dans un texte, d'ouvrir des paragraphes, de créer et gérer une bibliographie ou, encore, de spécifier des modifications typographiques. Adjoint à ces commandes, un *(dé)compilateur* transforme l'arbre abstrait en un texte d'entrée pour le formateur disponible sur le système opératoire (**compose** sur Multics, **nroff** sur UNIX).

Cette application de MENTOR se démarque nettement de celles dans lesquelles des textes écrits dans des langages formels sont édités. En effet, la syntaxe abstraite de Rapport est très simple mais les valeurs des terminaux génériques sont complexes et de taille importante (plusieurs lignes de texte); les langages formels ont souvent des caractéristiques inverses : la syntaxe abstraite est complexe et les terminaux génériques sont simples. Ceci a posé un défi car une des critiques majeures généralement faite à MENTOR concerne la médiocrité de l'interface utilisateur et l'impossibilité de modifier les terminaux génériques (il faut les recréer à chaque fois).

La réponse à ce défi consiste en l'inter-connexion de MENTOR et d'un éditeur de textes (**EMACS** et **TED** sur Multics ou **EMIN** sur UNIX). L'intérêt et l'originalité de cette solution sont principalement de dépasser l'opposition traditionnelle entre édition structurée et édition textuelle; l'utilisateur peut choisir à tout moment

le type d'éditeur qui lui semble le plus approprié. Les deux formes d'édition ne sont plus mutuellement exclusives mais coopérantes et complémentaires. Techniquement, ceci est réalisé par deux fonctions :

1. l'appel de l'éditeur de texte

   l'arbre pointé est *décompilé* et le texte obtenu est chargé dans le tampon de l'éditeur de textes qui est alors activé.

2. le retour à MENTOR

   le tampon est passé à l'analyseur syntaxique

   l'arbre abstrait correspondant est construit et greffé là où est le pointeur courant de MENTOR

Naturellement, les opérations précédentes sont effectuées par le système et sont, de ce fait, totalement transparentes à l'utilisateur. Cette coopération se traduit sur le terminal par un fenêtrage : la partie supérieure de l'écran est réservée à l'éditeur de texte et la partie inférieure à MENTOR.

La critique que l'on peut faire sur cette réalisation est l'existence de deux outils distincts qui utilisent chacun des concepts, des structures et des commandes différentes, ce qui oblige l'utilisateur a acquérir deux *savoir faire*.

8.1.3 MENTOR−TYPOL

L'utilisation des éditeurs syntaxiques (MENTOR, le Cornell Program Synthesizer (TEI,81) ou Gandalf (MED,82)) montre clairement que si le contrôle syntaxique des textes est une aide précieuse, il devient rapidement nécessaire d'intégrer des contrôles contextuels (contrôle de déclaration de variables, de types, ...). Cette intégration accroît considérablement l'efficacité de l'éditeur. Le travail de T. Despeyroux (DES,83) a donc consisté en l'étude et l'implantation d'un formalisme de spécification des contraintes contextuelles et la réalisation d'un mécanisme permettant le contrôle d'un texte.

Un des objectifs de cette étude est la création d'un générateur qui, comme METAL (KAH,83), autorise une définition statique des contraintes contextuelles qui est ensuite *compilée* pour produire les procédures de contrôle. La solution retenue est la suivante :

Les contraintes sont exprimées à l'aide de règles d'inférence écrites dans le formalisme TYPOL

Les règles sont ensuite traduites en un ensemble de clauses PROLOG et conservées comme base de règles

Le contrôle du texte est réalisé en deux étapes :

compilation du texte en une clause PROLOG

exécution, sous PROLOG, du programme constitué de la base de règles et de la clause précédente.

La critique que nous adressons à ce système concerne la lourdeur du mécanisme. En effet, la compilation du texte préalablement au contrôle et l'utilisation d'un système totalement extérieur à MENTOR en rendent peu probable une utilisation interactive. Néanmoins, TYPOL a le mérite d'exister et d'avoir été testé sur les langages ASPLE et ML. D'autres voies sont étudiées pour résoudre ce problème de la génération des contrôles, citons en particulier T. Reps (DEM,81) qui a défini un algorithme incrémental d'évaluation des attributs d'une grammaire.

8.1.4 Comparaison critique

Pour conclure, comparons EDME avec ces environnements sur les trois points importants que nous avons évoqués :

la multiplicité des commandes. Avec EDME, nous avons essayé d'apporter une réponse à ce problème en proposant à l'utilisateur une structure de *plus haut niveau* que l'arbre abstrait et en intégrant dans les commandes une partie du processus de construction. Il est certain qu'une évolution de MENTOR−PASCAL en ce sens (construction d'une couche englobant les commandes existantes) en ferait un outil extrêmement efficace.

la multiplicité des outils et des structures. Avec EDME, nous avons tenté de surmonter ce problème en utilisant un éditeur de texte sur une seule couche; de ce fait les deux structures ne se *recouvrent* pas mais *s'enchaînent* d'une manière claire.

les contrôles contextuels. La qualité principale des contrôles d'EDME est leur interactivité. Leur défaut majeur tient à leur réalisation (par des «fonctions sémantiques») qui rend très délicate et longue une modification de la syntaxe de MEDEE. L'existence d'un générateur de contrôles interactifs se révèle indispensable à une application comme EDME.

8.2 Le Programmer's Apprentice

Le **Programmer's Apprentice** (RIC,78) est un projet ambitieux qui vise deux objectifs : développer une méthode de programmation, les outils formels connexes et construire un *Assistant Programmeur* qui prenne en charge les tâches ingrates de la programmation.

La méthode de programmation repose sur une vue originale des programmes : un programme est vu comme un ensemble de *composants normalisés* interconnectés dont seule la fonction qu'ils remplissent est connue et dont on ne se soucie pas de la réalisation interne (ce sont des boites noires). Cette vision, et la méthode

qu'elle implique, est directement inspirée des techniques d'ingénierie utilisées par les électroniciens. La méthode de construction de programme peut alors être résumée ainsi :

- Identifier la fonction à réaliser et les composants nécessaires
- choisir et connecter les composants
- valider les connexions
- recommencer sur les composants qui ne réalisent pas une fonction élémentaire
- le processus s'arrête lorsque tous les composants sont élémentaires.

Notons tout de suite que cette méthode implique l'utilisation d'une importante bibliothèque de composants qui contient des composants élémentaires (i.e. dont l'utilisation est immédiate) et des composants *génériques* qui se composent d'un schéma de connexions et d'*emplacements* à compléter par d'autres composants.

Pour être effective, cette méthode nécessite de formaliser la notion de composant et celle de validation de connexion. Cette formalisation a été effectuée par C. RICH (RIC,81) qui appelle l'outil développé le *calcul de plans* (*plan calculus*).

La notion de composant est appelée *plan*. Un plan est caractérisé par :

- la fonction qu'il remplit
- des canaux d'entrées et de sorties
- des assertions sur les entrées et les sorties
- un flot de contrôle interne
- un flot de donnée interne
- des *méta variables* servant à désigner ses composants internes

Les trois dernières caractéristiques reflètent le fait qu'un plan peut être générique et représenter un *schéma* de calcul lorsqu'elles sont visibles à l'utilisateur, mais surtout, qu'un plan est une modélisation d'un programme à partir de laquelle on doit pouvoir générer un programme écrit dans un des langages de programmation habituel.

La validation des connexions est réalisée par la vérification des assertions d'entrée-sortie. Celles-ci sont calculées lorsque les méta-variables du plan ont été instanciées et donnent éventuellement lieu à une propagation dans le programme en cours de construction.

L'outil proposé, le *Programmer's Apprentice*, est un système qui doit coopérer avec le programmeur et faciliter son travail en effectuant les tâches annexes. Parmi celles-ci, citons la gestion du programme (ou plan) construit, la gestion de la bibliothèque de plans, le calcul des assertions et la vérification des contraintes, l'entretien de la documentation sur le travail en cours et la manipulation de celle du système.

L'apprenti a aussi des tâches plus intelligentes comme trier dans la bibliothèque les plans utilisables des autres, compte tenu des choix déjà réalisés, mais surtout il est chargé d'analyser les fragments de programme que l'utilisateur peut lui soumettre. En effet, il est prévu dans le Programmer's Apprentice que l'utilisateur puisse coder directement une fonction lorsqu'aucun plan lui convenant n'existe (pour un nouveau problème, pour un manque d'efficacité par exemple) ou lorsque cette façon de procéder lui semble plus simple. Le système analyse alors ce code pour le transformer en un plan, la seule structure interne manipulée, en construisant le plan de plus *haut* niveau possible. En cas d'échec, le système doit proposer le plan qui serait la *meilleure approximation* du code. Cette indication d'une approximation est intéressante car elle peut être la révélation d'une erreur de programmation. Un scénario d'interaction avec le Programmer's Apprentice dans lequel ce mécanisme apparaît est présenté dans (RIC,81), chapitre 2.

L'efficacité du Programmer's Apprentice dépend essentiellement de la bibliothèque de plans. En effet, si le programmeur a à sa disposition des plans correspondant aux fonctions à implanter, le programme sera très rapidement construit, sinon, il faudra assembler de multiples plans très élémentaires qui traduisent les structures fondamentales des langages de programmation. La question qui se pose alors concerne le contenu de cette bibliothèque, et corollairement, sa structuration.

Deux voies ont été suivies pour construire la bibliothèque. C. Rich a étudié le vocabulaire et les concepts habituellement employés. De cette étude théorique, il a déduit un ensemble de plans fréquemment utilisés. R. Waters (WAT,78) a adopté une approche plus pragmatique en analysant une bibliothèque mathématique de sous programmes écrits en FORTRAN. Pour effectuer ce travail, il a réalisé un système qui analyse un programme FORTRAN en en exhibant la structure logique sous forme de plans. Notons d'ailleurs à ce sujet une limitation pratique à ce type d'analyse : si les programmes bien structurés sont analysés sans peine, par contre l'analyseur est impuissant devant des programmes mal structurés et principalement devant ceux qui utilisent des effets de bords. Il est alors possible de définir un ensemble de plans. Le problème qui reste alors est de savoir si les bases obtenues sont générales où si elles sont spécifiques à un domaine particulier. Malheureusement, aucun indice ne permet de répondre, pas même la structuration de la base de plans.

La base de plans proposée dans (RIC,81) a l'allure générale d'un arbre (quoique quelques relations existent entre branches) : chaque noeud représente un plan dont les fils sont des *spécialisations* (i.e. des particularisations à une situation donnée). L'inconvénient principal que nous voyons à cette présentation est que les concepts concernant une technique de programmation et ceux concernant un domaine d'application n'apparaissent pas clairement : soit ils sont traduits en des plans distincts que l'organisation de la base mélange, soit les plans intègrent et mixent les deux types de connaissances. Nous pensons qu'un outil de programmation fondé sur une base de connaissances doit, pour être général, nettement séparer les connaissances sur les techniques fondamentales de programmation de celles sur le domaine car elles ne sont pas utilisées au même moment dans la construction d'un programme et chacune correspond à un type de décisions étrangères à l'autre. Notons que PSI (BAR,79), un système expert de génération de programmes, utilise une base de connaissance ainsi structurée.

Quoique les ambitions de Maiday soient beaucoup moins grandes que celles du Programmer's Apprentice, il nous semble possible de comparer les projets sur deux points :

- l'intégration d'une méthode aux outils
- les connaissances possédées par les outils.

La méthode retenue pour chacun des systèmes est différente, ce qui conduit à une conception différente des composantes de l'environnement; néanmoins, on retrouve les deux éléments essentiels qui sont l'existence d'un guide pour le programmeur et l'intégration des contrôles au processus de construction. La différence fondamentale nous semble concerner le type de réflexion que le programmeur a à réaliser à chaque étape de son travail. Avec Maiday, l'utilisateur doit identifier un objet (le nommer, le typer, le définir), la solution du problème (l'algorithme final) se construit peu à peu, sans qu'il y ait eu besoin de la concevoir au départ. Avec le Programmer's Apprentice, l'utilisateur doit, au contraire, fournir une solution pour réaliser une fonction à chaque étape (en choisissant un plan ou en identifiant les composants nécessaires). Or, imaginer une solution est une tâche ardue. De ce fait, le Programmer's Apprentice est plutôt destiné à des programmeurs experts alors que Maiday s'adresse aux novices.

Les connaissances possédées par les deux systèmes se différencient essentiellement par leur niveau : celles de Maiday concernent plutôt le processus de construction d'un programme alors que celles du Programmer's Apprentice concernent plutôt des *faits* de programmation. Nous pensons qu'ainsi, Maiday atteint à plus de généralité et qu'en particulier la question de la dépendance vis à vis d'un domaine ne se pose pas. Notre réalisation souffre la critique sur le point suivant : les connaissances de Maiday sont implicitement contenues dans le code des fonctions, ce qui est regrettable car elles sont alors difficilement extensibles.

8.3 CATY

CATY (GRE,84) (BEA,84) est un prototype de système de construction assistée de programmes développé au LRI. Il présente deux caractéristiques originales : l'emploi d'une méthode et d'outils formels dérivés des travaux sur la synthèse de programmes et un interface utilisateur fondé sur des représentations graphiques.

La méthode de programmation employée se décompose en deux étapes :

1. La traduction du problème :
   - identification de la variable résultat
   - identification du n-uple de variables données
   - nommage du programme

2. Le découpage du problème :
   - soit par un schéma de décomposition de programme
   - soit par un schéma de décomposition de données

L'identification des variables consiste essentiellement à leur associer un type abstrait algébrique. Les types sont généralement puisés dans une bibliothèque de types. L'apport original de la méthode concerne les schémas de décomposition de données : ceux-ci sont associés à un type (ou à un n-uple de types) et exprime la *structure algorithmique* des types, c'est à dire la structure intuitive des programmes les utilisant. Les programmes ainsi obtenus sont applicatifs, sans effet de bord et comportent généralement un (des) appel(s) récursif(s) au problème initial. Le lecteur intéressé trouvera une description plus détaillée de la méthode, des types et des schémas dans (GRE,84), chapitre IV.

Le système proposé est architecturé autour de la base de connaissance. Cette dernière contient des types abstraits et leur schémas de décomposition associés. Plusieurs modules gravitent autour de la base. Un système de manipulation de types permet d'en définir de nouveaux et de les vérifier grâce à un évaluateur symbolique. L'éditeur CATY qui gère le processus de construction des programmes en est bien évidemment le satellite principal. Enfin, une couche appelée *noyau* enveloppe les modules et gère l'interface utilisateur ainsi que certaines contraintes de cohérences statiques applicables au système. L'interaction avec l'utilisateur a été particulièrement étudiée. La couche *noyau* permet d'uniformiser les présentations et les commandes des différents modules. Techniquement, celle-ci est réalisée par des *extensions* écrites autour d'éditeurs pleine page (**EMACS** sur le système Multics et **WINNIE** sur UNIX); les commandes sont donc lancées grâce au mécanisme des *clés* (association d'un touche du clavier à une fonction) et les données visualisées à travers un écran multi-fenêtres. La partie la plus originale de l'interaction est la fenêtre graphique de CATY.

Cette possibilité graphique de CATY est rendue possible par l'emploi d'un poste de travail (une machine PERQ) constitué d'une unité de traitement autonome, d'un écran *bit-map* et d'une souris. La représentation graphique est identique pour les données (les schémas de décomposition) et les programmes; elle prend la forme d'un graphe étiqueté dont les noeuds sont des variables (représentées par des ellipses) et des opérateurs (opérations d'un type ou sous-problèmes, représentés par des rectangles). Cette réalisation nous semble intéressante à deux point de vues :

Sur le plan conceptuel. Le graphisme est considéré comme un bon support d'interaction, en particulier parce qu'il permet une visualisation et une compréhension rapide et intuitive du résultat. Ceci est d'autant plus intéressant avec CATY que la représentation choisie, très simple, favorise moins la visualisation d'un programme (comme les flôts de données ou de contrôle par exemple), que la visualisation de la structure du problème (variables et types utilisés, découpage et articulation des sous-problèmes).

Sur le plan architectural. CATY est réparti sur deux machines reliées par réseau; la gestion du processus de construction et celle de la base de connaissance sont effectuées sur une machine puissante (VAX 780) et l'interaction graphique est entièrement prise en charge par le poste de travail (machine PERQ). Cette séparation permet de tirer parti des caractéristiques de chaque machine : puissance de traitement et importante taille mémoire d'une part, souplesse et rapidité de l'interaction d'autre part. Cette solution présente aussi des inconvénients : l'un, d'ordre technique, concerne la définition et la gestion d'un protocole de communication relativement complexe, l'autre, beaucoup plus important, concerne la vérification et le maintien de la cohérence entre la base de connaissance et la base graphique. Ce dernier problème n'est actuellement pas résolu, la cohérence étant maintenue à la main.

Comme dans le cas du Programmer's Apprentice, précédemment étudié, l'efficacité de CATY dépend essentiellement de la base de connaissances. Si l'utilisation, désormais courante, des types abstraits permet d'imaginer aisément un contenu *réaliste* de la base de types (tant pour les types élémentaires (**entier, booléens,dates**, ...) que pour les types génériques (**liste, file, table**, ...)), il n'en est pas de même de la base des schémas de décomposition. La difficulté ne porte pas sur les schémas associés à un type unique, ceux-là sont souvent triviaux, mais sur les schémas associés à des n-uples de types qui correspondent souvent à une solution particulière d'un problème; or la plupart des applications réelles fournissent plusieurs résultats en consommant plusieurs types de données. La solution évoquée dans (GRE,84) consisterait à développer une base très importante contenant de nombreux *cas particuliers*. Deux questions se posent alors :

«Le programmeur ne risque-t-il pas d'être confronté à un problème de choix entre de trop nombreuses solutions ? »

«Que faire si aucun des schémas connus n'est applicable au problème précis à résoudre ? »

Il nous semble possible de comparer CATY et EDME sur plusieurs points :

- l'intégration d'une méthode au système
- l'interaction homme–machine
- les outils de développement des éditeurs

Il serait également possible de comparer les connaissances des deux systèmes, mais les conclusions seraient à peu près identiques à celles exposées lors de l'étude du Programmer's Apprentice.

L'intégration d'une méthode conduit, dans les deux systèmes, à concevoir l'élaboration d'un programme en deux étapes : premièrement, construire un algorithme sans se soucier de l'exécution (efficacité, ordre de construction, ...), ensuite, optimiser et traduire en un langage de programmation usuel. Dans les deux systèmes, il est donc envisagé d'adjoindre un module d'optimisation et de traduction. L'assistance fournie par les systèmes lors de la construction d'un programme est du même ordre; toutefois, CATY nous semble moins efficace qu'EDME dans le cas des situations bloquantes. Avec CATY, une telle situation peut arriver principalement dans deux cas : une inadaptation de la base de connaissances au problème traité (cf paragraphe précédent) ou une mauvaise traduction initiale du problème. Ce second point connu de ses

auteurs (cf (GRE,84), page 92) est directement lié à la méthode employée. Nous pensons que l'utilisation d'une méthode déductive avec EDME permet d'éviter ces blocages.

L'interaction homme−machine a été étudiée dans les deux systèmes et conduit à des solutions similaires, citons par exemple : l'utilisation du multi-fenêtrage, le guidage de l'utilisateur lors des commandes et le masquage de la structure interne du programme édité. L'utilisation d'un écran *bit-map*, d'une souris et d'une représentation graphique donne une qualité à l'interaction de CATY que ne possède pas encore EDME; un de nos projets les plus immédiats est l'adaptation de l'interface de Maiday à un poste de travail comparable (écran graphique et souris).

EDME et CATY possèdent deux caractéristiques communes : une représentation interne structurée du programme développé et un interface utilisant des techniques *video*. Les techniques de développement employées sont opposées : pour EDME, nous avons utilisé un éditeur structurel pré-existant (MENTOR) et programmé entièrement un interface, pour CATY, l'éditeur structurel spécifique a été programmé alors que l'interface utilise les mécanismes d'un éditeur *video* (EMACS ou WINNIE). Si l'approche de CATY nous semble intéressante en ce qui concerne l'indépendance vis à vis des terminaux (lorsque le graphisme n'est pas utilisé, évidemment), elle nous semble plus lourde que la nôtre en ce qui concerne l'effort de programmation (primitives de manipulation de la structure, analyseur syntaxique), les transports sur un autre système opératoire et l'évolution de la structure interne (modification de syntaxe).

8.4 bVLISP

Le système bVLISP (WER,83) est un environnement de programmation, développé par H. Wertz à l'Université de Paris VIII−Vincennes, destiné à faciliter la réalisation d'applications en LISP. Les trois qualités fondamentales de bVLISP sont : une haute interactivité, l'intégration de tous les outils et une assistance effective au programmeur. Notons enfin que bVLISP est actuellement opérationnel.

Les qualités précédemment citées résultent d'une analyse méthodologique et psychologique de la programmation interactive que nous résumons ainsi :

1. la programmation est perçue comme un va-et-vient entre communication et programmation, réalisé en collaboration avec un outil. La machine doit donc offrir des réponses immédiates qui sont soit le résultat des commandes, soit des erreurs détectées (syntaxiques, sémantiques, d'exécution, ...), soit de la documentation sur l'activité du moment ou le programme en cours de construction.
2. le développement d'un programme consiste de moins en moins en la conception d'un nouveau programme qu'en la modification d'un programme déjà existant. L'activité du programmeur est donc plutôt orientée vers la lecture, l'analyse, la compréhension, la modification et le test de fragments de code.

3. les difficultés de la programmation sont liées à nos faibles capacités de mémorisation *à court terme* alors que la construction d'un programme implique de manipuler simultanément de nombreux concepts, par exemple : le problème à résoudre, les données manipulées, le résultat souhaité, la solution retenue, la syntaxe et la sémantique du langage, le fragment de code, les commandes de l'outil, ...

Nous renvoyons le lecteur intéressé à (WER,83), chapitre 1 section 3 et chapitre 2, pour une analyse plus détaillée.

Les fonctionnalités de bVLISP peuvent être rangées en trois classes :

la documentation. Celle-ci se compose d'un manuel en-ligne qui a la particularité de pouvoir être consulté au milieu de la frappe d'une commande, d'un système appelé **menu** qui se compose d'un bandeau, affiché sur le dessus de l'écran, rappelant les commandes usuelles de l'outil en cours d'utilisation et d'un mécanisme de documentation incrémentale sur les fonctions construites (type de fonctions, références croisées des variables, ...).

l'assistance lors de l'exécution. La première forme d'assistance concerne les erreurs interrompant le calcul, pour certaines, bVLISP autorise une correction *à la volée* permettant de continuer le calcul. La seconde concerne l'observation de l'exécution, des outils classiques comme une fonction d'interruption programmable et une trace sont évidemment fournis ainsi qu'un outil de *trace video* qui dessine le graphe dynamique des appels de fonctions et positionne un curseur sur la fonction en cours d'activation. Une troisième forme se compose d'un ensemble d'outils de développement originaux, citons en particulier un mécanisme d'assertion qui permet de greffer des prédicats sur les valeurs d'entrée ou de sortie d'une fonction qui sont évalués à chaque appel et un mécanisme d'exemples qui permet de greffer à une fonction une liste de couples (valeur des paramètres, valeur souhaitée du résultat). Au premier appel d'une fonction munie d'exemples, bVLISP applique la fonction aux paramètres et compare les résultats obtenus et souhaités, l'utilisateur est informé lorsque les deux résultats divergent.

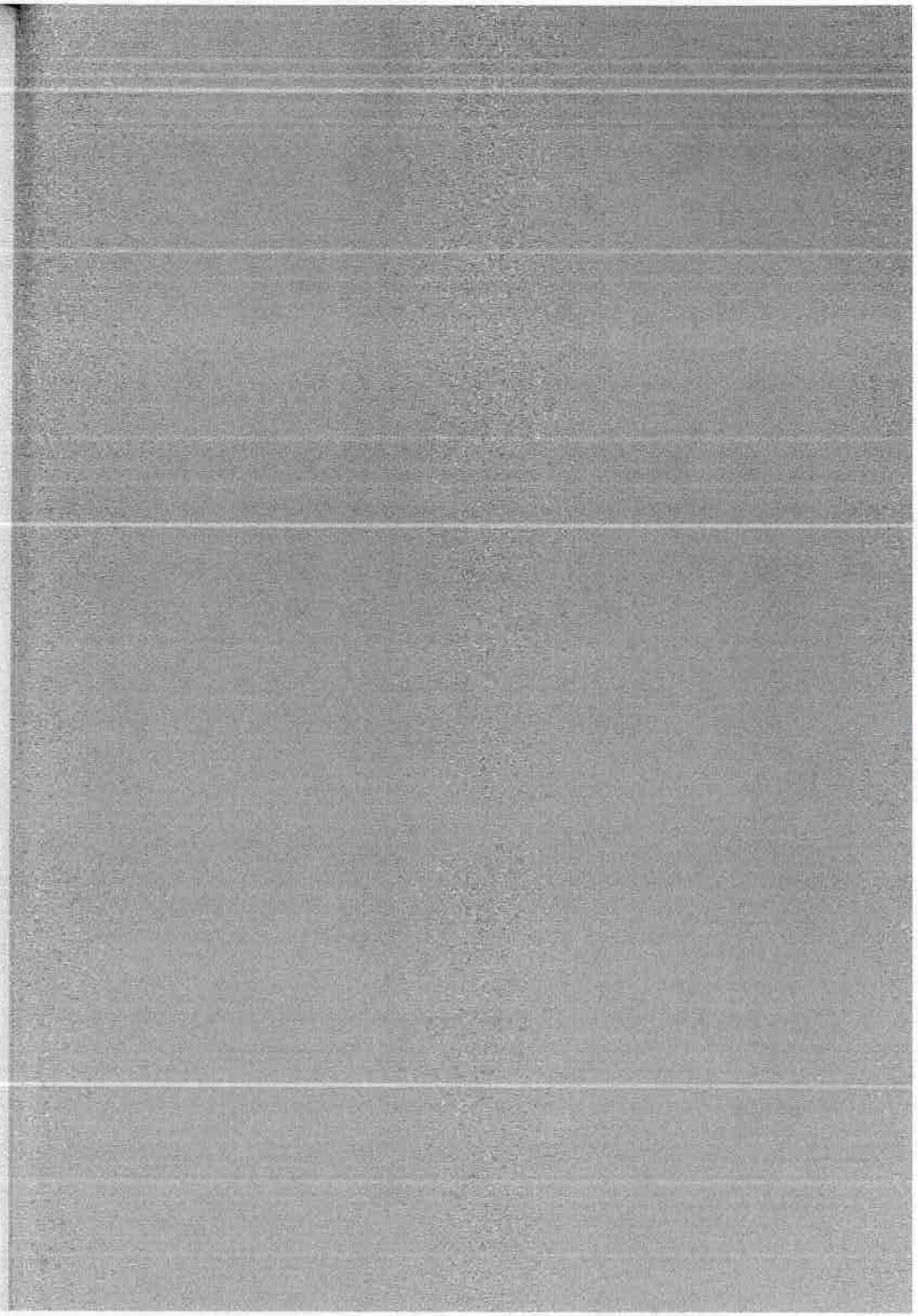
l'éditeur. L'éditeur se compose de deux parties : un éditeur structurel LISP classique et un gestionnaire de versions. En effet, chaque modification d'une fonction correspond à l'élaboration d'une nouvelle version de la fonction. bVLISP conserve toutes les versions qui peuvent être rééditées à tout moment.

L'intégration des outils peut être située sur trois plans :

1. le plan de l'interaction avec l'utilisateur. L'interface utilisateur est uniforme pour tous les outils. L'écran typique est séparé en trois fenêtres : soit, du haut vers le bas, le bandeau du menu, la fenêtre d'affichage des résultats et la *fenêtre d'interaction* dans laquelle l'utilisateur rentre les commandes et les données. La syntaxe des commandes est celle des fonctions LISP.
2. le plan des activités. Lorsqu'une commande est lancée, bVLISP déclenche automatiquement toutes les activités connexes (gestion des versions avec l'éditeur ou évaluation des assertions et exemples avec l'exécution, par exemple) sans que l'utilisateur en ait conscience. Seuls les cas pour lesquels apparaît un *problème* (un exemple non vérifié, par exemple) sont signalés au programmeur.

3. le plan des objets manipulés. Il nous semble essentiel de noter que malgré la diversité des outils et des concepts, bVLISP ne manipule qu'une structure unique qui intègre toute l'information. Ce point nous semble très intéressant car il montre qu'un programme ne doit pas être considéré sous le seul point de vue d'un fragment de code mais plutôt comme un ensemble de points de vue : une histoire, des spécifications (exemples, assertions), des commentaires, une documentation, ...

Maiday et bVLISP présentent deux approches complémentaires de la construction de programmes. bVLISP, en insistant sur les aspects d'exécution et de lectures d'un programme, favorise une démarche *expérimentale*, par essais-erreurs, Maiday favorise plutôt une démarche planifiée. Ces deux démarches correspondent à des moments différents de la programmation : la première est plutôt adaptée à la mise au point, à l'optimisation ou à l'adaptation à un problème proche d'une solution existante, la seconde est plus adaptée à la découverte d'une première solution. Il nous paraît certain que Maiday, pour devenir opérationnel, devra intégrer des outils sophistiqués liés à l'exécution inspirés de ceux de bVLISP.



Chapitre 9

Conclusion

Nous avons abordé dans ce travail trois domaines d'étude pour lesquels Maiday apporte des enseignements:

l'aspect méthodologique. L'apport de Maiday est ici de montrer qu'un outil *réaliste* (i.e. auquel on ne demande pas «trop» d'intelligence et qui soit relativement efficace) peut intégrer une méthode. Cette dernière s'est révélée être par ailleurs un bon support pour définir les fonctionnalités d'un système. Notre effort a principalement porté sur la construction d'une solution, il faut maintenant travailler à l'intégration de l'aspect des réutilisations d'algorithmes ou de *schémas* de solution. L'absence de ce dernier aspect dans Maiday est probablement le point le plus frustrant du système: les deux démarches, construction et réutilisation, sont complémentaires.

l'aspect ergonomique. La réflexion sur les structures perçues par le programmeur, l'assistance et les communications entre l'homme et la machine nous ont permis de simplifier au maximum l'interaction, et en particulier de proposer un nombre très restreint de commandes. Maiday montre ici que cette approche s'avère réaliste, car on peut créer réellement des algorithmes avec la maquette actuelle, et très efficace sur le plan de l'apprentissage. Il montre aussi que la conception d'un bon interface utilisateur doit être conjointe à la définition des fonctionnalités du système.

l'aspect d'ingénierie du logiciel. En choisissant de construire Maiday entièrement sur MENTOR, nous prenions deux risques: celui d'obtenir un système très inefficace (trop lent pour être vraiment interactif par exemple) et surtout celui de nous heurter rapidement à des limitations de MENTOR. En contre-partie, nous espérions obtenir un développement rapide. Notre réalisation montre que le pari a été gagné.

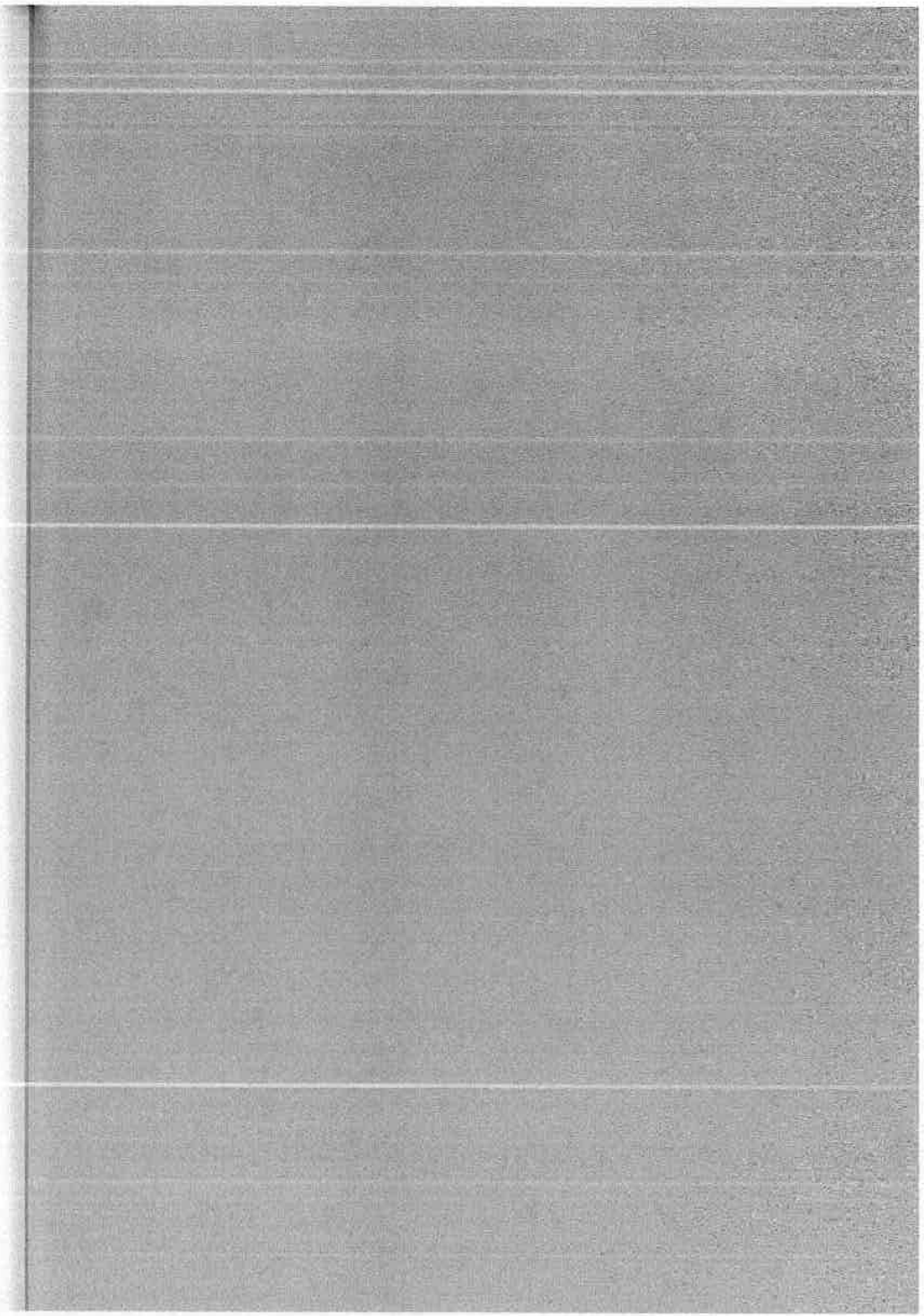
La démarche expérimentale que nous avons suivie nous a déjà amené à proposer des améliorations à Maiday (en particulier sur EDME) lors de la critique de notre réalisation. En plus de ces extensions immédiates, d'autres travaux peuvent utiliser l'expérience acquise. En particulier, une voie de recherche prometteuse nous semble être l'extension de la réalisation actuelle par un paramétrage des fonctionnalités.

Les possibilités de paramétrer Maiday sont nombreuses. Nous avons déjà évoqué le paramétrage de l'interface utilisateur, soit en ce qui concerne le formatage de textes, soit en ce qui concerne le nombre et la disposition des fenêtres. Les problèmes sont ici principalement d'ordre technique et commencent à être maîtrisés. Nous voudrions évoquer les deux paramètres qu'il nous semble essentiel d'étudier pour augmenter la puissance de Maiday: les stratégies de construction et les méthodes de programmation.

L'intégration de ces deux paramètres nécessite encore d'importants travaux théoriques. En ce qui concerne les stratégies de construction, il faut les recenser, les identifier et étudier leur pertinence vis à vis des types d'utilisateurs, des types de problèmes, des domaines d'application, ... Partant de ces résultats, il deviendra alors possible de déterminer des critères de choix d'une stratégie pour un problème et un utilisateur donné qui permettront éventuellement à Maiday de suggérer au programmeur un *plan de travail*. Notons que la

première partie de ces travaux est actuellement abordée par J.M. Hoc qui utilise Maiday. En ce qui concerne les méthodes, il convient de formaliser les différentes méthodes et essentiellement de les intégrer dans un cadre unique; ensuite, il conviendra de préciser les critères de choix associés à chacune d'elles en fonction des types de problèmes ou de l'expérience du programmeur, par exemple.

Lorsque ces deux problèmes auront été résolus, il sera possible de concevoir l'outil ultime de la programmation: celui qui construira seul la solution à partir des choix d'analyse que lui aura communiqué le programmeur.



REFERENCES

(ADA,80) ADA
manuel de référence
CII−HB
1980

(AND,84) E. ANDRE, B. MOREAU, B. ROUGEOT
Vers un Atelier Flexible et Intégré de Logiciel: le Projet CONCERTO
L'Echo de Recherches, n[115
1984

(ASH,77) E.A. ASHCROFT, W.W. WADGE
LUCID, a non procedural language with iteration
Communication of the ACM, vol 24, n[9, pp 563−573
Septembre, 81

(AYE,84) B. EL AYEB
Une boite à outils pour un environnement de spécification
Mémoire de Maitrise − Facultés Universitaires Notre Dame de la Paix
Namur − Belgique
Septembre 84

(BAL,77) R. BALZER, N. GOLDMAN, D. WHILE
On the use of programming knowledge to understand informal process description
Information Science Institute report ISI/RR−77−63, University of South
California, Marina del Rey, CA
October 77

(BAR,79) D.R. BARSTOW
Knowledge Based Program Construction
Programming Langages Series − The Computer Science Library − North Holland
1979

(BEA,84) M. BEAUDOIN−LAFON, C. GRESSE
CATY: un environnement de programmation pour une construction graphique et interactive de programmes
2ème Colloque de Génie Logiciel − Nice − pp 313−328
Juin, 84

(BEL,78) F. BELLEGARDE, J.P. FINANCE, B. HUC, J. JARAY, P. LESCANNE, J. MAROLDT, C. PAIR, A. QUERE, J.L. REMY
A type of language for the deductive programming method
Conference on Reliable Software − German ACM Chapter − Bonn
1978

(BEL,84) F. BELLEGARDE
Rewriting Systems on FP Expressions that Reduce the Intermediate Sequences they Yield
Proc. LISP and Functional Programming − Austin − Texas
August, 84

(BUY,83) M. BUYSE, P. VANHEMELRYCK
Création d'un environnement de spécification pour le langage SPES
Mémoire de Maitrise – Facultés Universitaires Notre Dame de la Paix – Namur – Belgique
Septembre, 83

(COL,83) A. COLMERAUER, H. KANOUI, M VAN CANEGHEM
Prolog, bases théoriques et développements actuels
TSI – Vol 2, n]4
Juillet, 83

(COY,82) H. COYOTE
Rapport de DEA – non publié – CRIN – Nancy
1982

(DES,84) T. DESPEYROUX *Executable Specification of Static Semantic*
Rapport de recherche n[295 – INRIA
Mai,84

(DUB,84) E. DUBOIS
Cadre et méthode de spécification de systèmes d'information fondés sur les types de données
Thèse de Docteur Ingénieur – INPL
Mars, 84

(DUC,84) A. DUCRIN
Programmation : – du problème à l'algorithme (tome 1)
– de l'algorithme au programme (tome 2)
Dunod ed.
Septembre, 984

(DIJ,76) E.W. DIJSKTRA
A Discipline of Programming
Englewood Cliffs – Prentice Hall
1977

(DON,75) V. DONZEAU–GOUGE, G. KAHN, G. HUET, B. LANG, J.J. LEVY
A structure oriented program editor: a first step towards computer assisted programming
International Computing Symposium – North Holland Publishing Co
1975

(DON,80) V. DONZEAU–GOUGE, G. HUET, G. KAHN, B. LANG
Programming Environments Based on Structure Editors: the MENTOR Experience
Workshop on Programming Environments in Ridge Field, LT
June, 80

(EST,83) J. ESTUBLIER, S. KRAKOWIAK, J. MOSSIERE, Y. ROUZAUD
Design principles of the ADELE programming environment
Proc. Intenational Computing Symposium on Application Systems Development (ACM) – Nuremberg – March, 83

(EST,84) J. ESTUBLIER, S. GHOUL
Un système automatique des gestion de gros logiciels: la base de programmes ADELE
Proc. 2ème Congrès de Génie Logiciel – Nice – pp 267–278
Juin, 84

(FIN,79) J.P. FINANCE
Etude de la Construction des Programmes: Méthodes et Langages de Spécification et de Résolution de Problèmes
Thèse d'Etat – Université de Nancy I
Octobre, 79

(FIN,84a) J.P. FINANCE, M. GRANDBASTIEN, N. LEVY, A. QUERE, J. SOUQUIERE
SPES: un système pour spécifier et transformer
Proc. 2ème Congrés de Génie Logiciel – Nice – pp 345–357
Juin, 84

FIN,84b) J.P. FINANCE, J. SOUQUIERE
A method and a language for constructing iterative programs
à paraitre dans Science of Computer Programming

(GOL,80) A. GOLBERG,
SMALLTALK–80, The language and its implementation
Addison–Wesley Publishing Company
1983

(GRE,84) C. GRESSE
Contribution à la programmation automatique. CATY: un système de construction assitée de programmes
Thèse d'Etat, Université de Paris Sud – LRI (Orsay)
Mars, 84

(GUI,80) G. GUIHO, C. GRESSE, M. BIDOIT
Conception et certification des programmes à partir d'une décomposition par les données
R.A.I.R.O. Informatique, vol 14, n[4
1980

(GUY,80) J. GUYARD, P. LESCANNE
Une initiation à MENTOR
Rapport CRIN 80–R–04
1980

(G&W,84) P. GREUSSAY, H. WERTZ
Outils de développement pour la mise au point et de lecture de programmes LISP
Rapport final, contrat ATP 81.9B.403 – LITP
Mars, 84

(HER,84) N. HERSTSCHUH, P. BERGER
MAPPER, un exemple de «langage» nouvelle génération
Technique et Science Informatiques, vol 3, n[2
Mars, 84

(HOC,83) J.M. HOC
Une méthode de classification préalable des problèmes d'un domaine pour l'analyse des stratégies de résolution
Le Travail Humain, vol 46, pp 205−217
1983

(HOC,84) J.M. HOC, J. GUYARD, M. QUERE, J.P. JACQUOT
Designing and evaluating computer aids in structured programming
Proc. INTERACT − London
Septembre, 84

(HUC,78) B. HUC
Mise en oeuvre de la méthode de programmation déductive
Thèse de 3ème cycle INP − CRIN − Nancy
1978

(HUL,83) J.M. HULLOT
CEYX, a multiformalism programming environment
Proc IFIP − Paris − pp 223−227
1983

(HUL,84) J.M. HULLOT
CEYX version 4 : Le Manuel de Référence (brouillon)
INRIA
Janvier, 84

(JAC,82) J.P. JACQUOT
Réalisation du système d'affichage de l'éditeur méthodologique MEDEDIT
Rapport de DEA − CRIN − 82−R−084
Septembre, 82

(JON,82) C.B. JONES
Software Development, A Rigorous Approach
International Series in Computer Science, C.A.R. Hoare, ed. − Prentice Hall
1982

(KAH,83) G. KAHN, B. LANG, B. MELESE, E. MORCOS
METAL: A formalism to specify formalisms
Science of Computer Programming, vol 3, n[2
August, 83

(KOL,79) E. KOLMAYER
Evaluation de la méthode déductive de programmation
rapport CRIN 79−R−074
1979

(LAM,81) A. VAN LAMSWEERDE
Quelques approche à la production automatisée de logiciels d'application
Actes de la Journée de Synthèse, congrès AFCET Informatique − Gif sur Yvette
Novembre, 81

(LAM,83) A. VAN LAMSWEERDE
Automatisation de la production de logiciels d'application: quelques approches
Technique et Science Informatiques, vol 1, n[6 et vol 2, n[1
Novembre 82, Janvier 83

(LEC,84) C. LE CRAS
Rapport de DEA − CRIN
Septembre, 84

(MED,82) R. MEDINA−MORA
Syntax−Directed Editing: Towards Integrated Programming Environments
Ph. D. Thesis. − Carnegie Mellon University − Dep. of Computer Science
March, 82

(MEL,81) B. MELESE
Manipulation de programmes PASCAL au niveau des concepts du langage
Thèse de 3ème cycle− Université de Paris XI Orsay
Juin, 80

(MEL,82) B. MELESE
Manuel d'utilisation de METAL − version 1
non publié − INRIA
1982

(MEL,84) B. MELESE
Edition structurée − Edition non structurée − Coopération et complémentarité
Proc 2ème congrès de Génie Logiciel − Nice − pp 123−140
Juin, 84

(M&VD,82) N. MEYROWITZ, A. VAN DAM
Interactive Editing Systems
Computing Surveys, vol 14, n[3, pp 321−415
September, 82

(PAI,79) C. PAIR
La construction des programmes
R.A.I.RO. Informatique, vol 13, n[2
1979

(REP,84) T. REPS, T. TEITELBAUM, A. DEMERS
Incremental Context!-Dependent Analysis for Language!-Based Editors
Procs. Cours INRIA «Les éditeurs dirigés par la syntaxe» AUSSOIS
18−22 Avril 1984

(RIC,78) C. RICH, H.E SHROBE, R.C. WATERS, G.J. SUSSMAN, C.E. HEWITT
Programming viewed as an engineering activity
M.I.T. − Artificial Intelligence Laboratory − AI.memo 459
January, 78

(RIC,81) C. RICH
Inspection Methods in Programming

Ph D Thesis, M.I.T. – Artificial Intelligence Laboratory
June, 81

(SCH,79) P.C. SCHOLL
Vers une programmation systèmatique: Etude de quelques mèthodes, techniques et outils
Thèse d'Etat – USMG et INPG – Grenoble
1979

(SCH,83) W. SCHERLIS, D. SCOTT
First steps towards inferential programming
Proc. IFIP, pp 199–212
1983

(SOU,82) J. SOUQUIERE
Construction et transformation de programmes itératifs
Thèse de Docteur Ingénieur – Université de Nancy I
Mars, 82

(SOU,83) L. SOUFI
Un système de construction de programmes
Proc Journées BIGRE 83 – Le Cap d'Agde – pp204–224
Octobre, 84

(TEI,81) T. TEITELBAUM, T REPS
The Cornell Program Synthesizer: A syntax directed programming environment
Communication of the ACM, vol 24, n[9, pp 563–573
September, 81

(TEI,78) W. TEITELMAN
INTERLISP Reference Manual
Xerox Palo Alto Research Center, Palo Alto
October, 78

(WAR,79) D. WARNIER
Les procédures de traitement et leur données (LCP)
Editions d'Organisation – PARIS
1979

(WAT,78) R. WATERS
Automatic Analysis of the Logical Structure of Programs
Ph. D Thesis, M.I.T. – Atificial Intelligence Laboratory
December, 78

(WEI,71) G.M. WEINBERG
The Psychology of Computer Programming
Van Nostrand–Reinhold (New–York)
1971

(WER,84) H. WERTZ
Etude, réalisation et évaluation d'un environnement de programmation utilisant des représentations multiples pour le développement continu de logiciels trés évolués

Thèse d'Etat – Univecrsité de Paris VIII – Vincennes
Novembre 83

(ZAC,82) J. ZACHARY
A Syntax–Directed Tool for Constructing Specifications
Master of Computer Science thesis – M.I.T.
March, 83

(ZAC,83) J. ZACHARY
Type inference in MEDEE
Rapport CRIN 83–R–043
Septembre, 83

(ZER,84) L. ZERTAL
Inférence de types dans le langage MEDEE
Rapport de DEA – CRIN
Septembre, 84

institut
national
polytechnique
de lorraine

Le Président,

N/Réf. : Scol.

AUTORISATION DE SOUTENANCE DE THESE DE DOCTORAT-D'INGENIEUR

-:-:-:-:-:-:-:-:-:-:-:-:-:-:-

VU LE RAPPORT ETABLI PAR :

Monsieur le Professeur FINANCE J-P.

le Président de l'Institut National Polytechnique de Lorraine autorise :

Monsieur JACQUOT Jean-Pierre

à soutenir, devant l'I.N.P.L., une thèse intitulée :

"ETUDE D'UN OUTIL D'ASSISTANCE A LA CONCEPTION METHODIQUE DE PROGRAMMES. REALISATION ET EVALUATION D'UNE MAQUETTE"

en vue de l'obtention du titre de DOCTEUR-INGENIEUR

Spécialité *"INFORMATIQUE"*

Fait à NANCY, le *7 Septembre 1984*

P/O Le Président de l'I.N.P.L.
Le Vice-Président du Conseil Scientifique
de l'I.N.P.L.

M. LUCIUS
A. MAILFERT

RESUME

Le travail présenté dans cette thèse s'inscrit dans le courant actuel du génie logiciel et plus particulièrement dans la branche traitant des environnements de programmation. Les dernières années ont vu l'apparition d'outils sophistiqués de manipulation de programmes: les éditeurs dirigés par la syntaxe des langages utilisés. Les contrôles que de tels systèmes permettent de faire sur un programme sont souvent encore limités à la syntaxe. Pour être réellement utiles et efficaces, ils doivent aussi porter sur la sémantique statique et la méthode suivie par l'utilisateur.

Cette thèse est la relation d'une expérience dont l'objet est la construction d'un environnement de programmation original : MAIDAY. L'originalité du travail réside dans la focalisation sur le processus de construction et l'utilisation intelligente de méta−outils tels que MENTOR.